NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DESIGN AND ANALYSIS OF A STYLIZED NATURAL GRAMMAR FOR AN OBJECT-ORIENTED LANGUAGE (OMEGA)

by

Robert P. Ufford

June 1985

Thesis Advisor: B. J. MacLennan

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE *(and Subtitle)*

The Design and Analysis of a Stylized Natual Grammar for an Object-Oriented Language (Omega)

5. TYPE OF REPORT & PERIOD COVERED

Master's Thesis
June 1985

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Robert P. Ufford

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Naval Postgraduate School
Monterey, CA 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Postgraduate School
Monterey, CA 93943

12. REPORT DATE

June 1985

13. NUMBER OF PAGES

118

14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*

15. SECURITY CLASS. *(of this report)*

UNCLASSIFIED

15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

object-oriented language, grammars, natural syntax, rule-based paradigm, Omega

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

In this thesis we discuss the design issues of a stylized natural language syntax for Omega, an object-oriented programming language built upon rule-based pattern matching. Emphasis is placed on simplicity and flexibility in the design. The feasibility of the proposed grammar (Omega-1.5) is evaluated by developing a prototype translator to translate the Omega-1.5 grammar into the predicate logic style of Omega-1. Sample applications are (cont)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ABSTRACT (Continued)

provided to examine the features of the grammar and to explore possible application areas. Limitations in the design are analyzed and potential amelioriations are suggested. We conludé with a general assessment of the overall Omega system.

Approved for public release; distribution is unlimited.

The Design and Analysis of a Stylized Natural Grammar for an Object-Oriented Language (Omega)

by

Robert P. Ufford
Captain, United States Army
B.S., United States Military Academy, 1978

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

ABSTRACT

In this thesis we discuss the design issues of a stylized natural language syntax for Omega, an object-oriented programming language built upon rule-based pattern matching. Emphasis is placed on simplicity and flexibility in the design. The feasibility of the proposed grammar (Omega-1.5) is evaluated by developing a prototype translator to translate the Omega-1.5 grammar into the predicate logic style of Omega-1. Sample applications are provided to examine the features of the grammar and to explore possible application areas. Limitations in the design are analyzed and potential amelioriations are suggested. We conclude with a general assessment of the overall Omega system.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

The evolving complexity of modern computer applications is leading to basic changes in the nature of programming. There is a growing awareness that conventional programming languages are not adequate for building computer systems. Programmers are demanding increasingly sophisticated tools for understanding and manipulating intricate, ill-defined problem domains. Successive conventional languages have had little success in providing additional tools to help the programmer combat the complexity barriers. Although the languages are getting larger, they are not getting stronger. As John Backus stated, "Inherent defects at the most basic level cause them to be fat and weak...." [Ref. 1: p. 613]

Backus further stated that a major limitation of the conventional languages was the "word-at-a-time" programming style. An example of this style is evidenced in the array construct [Ref. 2: p. 404]. Arrays are processed by performing an action on each individual element, with all of the indexing and loop control that this action requires. Thus, the programmer is occupied with minute implementation details rather than confining his thinking to the larger conceptual units of the task.

Programmers must shift their focus away from the detailed specifications of algorithms. The basic use of programming systems is not in developing sequences of instructions for accomplishing tasks, but in expressing and controlling descriptions of computational processes [Ref. 3: p.393]. High level languages were initially developed to free the programmer from the burdensome details of machine

code. Languages with even higher levels of abstractions are now required to rescue the programmer from inundation by unnecessary implementation-related details. Increased semantic power from the use of abstraction cannot be achieved, however, at the expense of architectural effectiveness. The conventional notion of programming languages needs to be reevaluated.

Alternatives to conventional languages have existed for quite some time. An early example is LISP. The original LISP system was characterized by the application of pure functions to list structures. This application of a function tc its argument is indicative of applicative programming. Other alternatives tc conventional languages are cbject-oriented programming and logic programming. One language framework that combines the features of the applicative, object-oriented, and logic programming categories is a language called Omega [Ref. 4]. This thesis shall focus on the features of the Omega framework.

B. THE CMEGA LANGUAGE

In order to understand the foundations of Omega, it is necessary to analyze the three categories of alternative languages mentioned ir section A (see sections C, D, and E). The influences upon Cmega from languages in these categcries will become quite cbvious as the features of Omega are explored. First, however, a general overview of Omega is in order. The backbone of Omega is the concept of object-oriented programming. A pioneer language in the object-oriented field was Simula [Ref. 5]. As the name suggests, Simula views all programming as simulation. This concept is fundamental to Omega's view of objects.

One unique feature of Omega is the provision for four alternative syntactic forms which represent the same

language. The first form (Omega-1) uses a predicate logic style and is the easiest to parse. The second and third forms use syntactic "tricks" to approach a pseudo-natural language format. This style is much easier for a naive computer user to read. Omega-4 further addresses the readability issue by using a two-dimensional format built upon the use of a form. The notion of multiple syntaxes creates a rich environment that supports many levels of user sophistication.

C. OBJECT-ORIENTED LANGUAGES

The object-oriented paradigm of Simula was smoothed and cemented in the Smalltalk language [Ref. 6]. It was the Smalltalk programming system that actually produced the term "object-oriented." Although there is some evidence of LISP in Smalltalk, the class notion from Simula has become dominant in the design. The class notion is the basic structural unit, with instances of classes, or objects, being the concrete units which comprise the Smalltalk system.

There are many advantages in the object-oriented approach. The simulation paradigm of objects is well suited to modeling real-world objects. Another advantage is the concept of state--objects hold the state of a computation. Additionally, an object orientation easily supports such concepts as abstract data types, information hiding, and modularization. A more intuitive appeal of objects is simply a sense of uniformity. No object is given any special status; there are no "second class citizens." A user-defined object is an object just like a system-defined object.

D. APPLICATIVE LANGUAGES

Applicative programming extends the model of mathematics to the world of computer programming. Applicative languages basically involve the application of functions to their arguments. Underneath the various syntactic idiosyncrasies of applicative languages is the rigorous structure of the lambda calculus. Various syntactical forms are merely "syntactic sugar" [Ref. 7] to help soften the rigid appearance of the lambda calculus format. Two well known applicative languages are pure LISP [Ref. 8] and the FP language [Ref. 1].

Applicative programming encourages the use of higher levels of abstraction through the use of functionals. Functionals are mechanisms for modifying the behavior of existing programs by combining primitive computational units into complex, powerful collections. Specifically, a functional is a function which receives functions as arguments and returns functions as results.

Another advantage of applicative programming is the notion of manifest interfaces. That is, the input-output connections to a subexpression are distinct and there are no hidden interfaces to complicate the semantics of a process. A final benefit is parallel evaluation, which is supported by the evaluation order independence of expressions.

Applicative language programming is essentially synonymous with value-oriented programming. Consequently, it is subject to the basic characteristics that are associated with values. The notions of time and state are lacking in value-oriented programming. This limits applications where temporal relationships are required.

E. LOGIC PROGRAMMING AND INFERENCE SYSTEMS

The development of Prolog [Ref. 9] in 1970 has made logic programming quite popular in recent years. Prolog has many applications in the artificial intelligence and inferential programming fields. It has been selected by the Japanese as the core language for their much-touted Fifth Generation Project [Ref. 10]. Prolog uses rule-based pattern matching as the basis for computation. A Prolog program consists of clauses, where each clause is either a fact or a rule about how the solution may be "inferred" from the database of facts. This is the first step toward logic programming. In conventional languages, different formalisms are used for expressing programs, databases, specifications, and constraints. Logic can be used to provide a single uniform language for all of these tasks.

Inference systems are usually associated with artificial intelligence applications. Rule-based paradigms have been used for problem-solving production systems and even for knowledge representation. A popular application has been in expert systems. For example, MYCIN [Ref. 11] and INTERNIST [Ref. 12] are two well-known systems in the medical field. XCON [Ref. 13], another example, is a system used at Carnegie-Mellon University for configuring computer components.

F. DEVELOPING AN ALTERNATIVE SYNTAX

The objective of this thesis is to develop and implement a natural language style syntax for the Omega language. Concepts from the Omega-2 and Omega-3 syntaxes will be synthesized into the development of an Omega-1.5 grammar. The ideal engineering solution for Omega-1.5 is to create a highly readable syntax at a minimal cost. Flexibility, as well as simplicity, is key to the design.

The feasibility of the 1.5 grammar was demonstrated by constructing a translator to translate programs written in the 1.5 grammar into the predicate logic syntax of Omega-1. The development of the translator was considered to be a learning process in that language features of the 1.5 grammar were studied and changed as necessary throughout the programming process. Translator features such as efficient code generation and elaborate error checking were considered to be of secondary importance.

Another objective of the thesis was to develop example applications in the 1.5 grammar and to run them first on the translator and then to run the translation on the McArthur interpreter [Ref. 14]. This approach permits an informal evaluation of the naturalized syntax. Additionally, it provides a beneficial vehicle for evaluating potential application areas for the Omega-1.5 grammar.

The Omega language is still in the experimental stages. Therefore, some attention has been placed on the general features of the language. Possible future design modifications are suggested and subjectively evaluated. Deviations from language features of the McArthur prototype are also noted. A final task is an introspective evaluation of Omega as a general-purpose programming language.

II. THE OMEGA-1 IMPLEMENTATION

A. PREFACE

General features of the Omega-1 syntax will be discussed in this chapter. The Omega concept was originally developed by Bruce MacLennan. A description of the language and a formal syntax for these constructs is presented in [Ref. 4]. These constructs were implemented by McArthur [Ref. 14] through a prototype interpreter. Some semantic and syntactic differences do exist between the original theory of the language and the actual implementation. A listing of these differences can be found in [Ref. 15]. The following summary will discuss Omega as amended by the prototype implementation.

B. OBJECTS AND VALUES

The basic elements in Omega are values and objects. A detailed discussion of the two is in [Ref. 16]. Briefly, objects are entities that have a unique identity and possess the following characteristics:

- objects exist in time and can change in time.
- objects may be created and destroyed.
- objects are unique, but may be shared.
- objects have a state (the sum of the relationships with all other objects in the system).

Values are mathematical entities and thus have the following characteristics:

- values exist independently of time.
- values are not subject to change.
- values cannot be created or destroyed.

Typical values in Omega include character strings, integers, and lists. A list is a collection of expressions enclosed by brackets. Two examples of lists are:

```
[red,white,blue]
[1,2,[1,2]]
```

C. RELATIONS

Relations are the "glue" which connect the components in Omega. In mathematical terms, R is a relation on the n sets, $s_1, s_2, \ldots s_n$, if R is a subset of the cartesian product $S_1 \times S_2 \times \ldots S_n$. Informally, a relation is a set of tuples, which are simply ordered collections of objects and values. Unlike relational database models, named attributes are not used to describe tuples. Instead, elements of tuples can be described by value, by relative position, and by pattern-matching. Tuples in a relation are unique. Additionally, there is no order among the tuples in a relation.

Relations are described either through pattern-matching or by name. A possible relation in Omega is:

```
perform(compilers,[scanning,parsing,
                                code_generation])
```

This relation is named by the identifier **perform**. It consists of a binary tuple, **<compiler,[scanning, parsing, code_generation]>**, that contains the object **compiler** and a list of the objects **scanning, parsing,** and code_generation. It should be noted that relations (and objects) in Omega

must be defined prior to their use. Definitions are established through procedure calls (section F).

Relations help determine the state of an object. An objects's state is defined by its associations with other values and objects in each of the relations in which it is a member. Relations are also objects, although they differ in that they have the inherent value of their tuples. As objects, relations may participate in other relations as a member of a tuple.

D. THE PRODUCTION RULE SYSTEM

The behavior of the entities in Omega is described through pattern-directed production rules. Through these rules, state transitions in the system can be described. Rules are written as implications of the form:

if <premise> -> <conclusion>

The premise consists of one or more boolean conditions pertaining to the state of the system. The conclusion defines actions to be taken whenever the conditions of the premise are true.

Inquiries and constraints are two of the basic constructs in the premise condition. Inquiries are expressed as:

if P(x,y,z) -> ...

Here we are testing to see if there exists (existential quantification) a tuple **<x,y,z>** in relation P. The meaning of the premise depends upon the bindings of x, y, and z. If we assume that these are unbound variables, then P(x,y,z) will match any ternary tuple in relation P. The match will result in the binding of the tuple's components to the variables x, y, and z.

A more complex inquiry might be:

```
if P(x,y,z), Q(f,y,g) -> ...
```

The comma between the two conditions denotes the logical conjunction of the two conditions in the inquiry. Thus in order for the premise to be true, the relations P and Q must each have a triple such that the second component of each triple is the same. When this condition occurs, the rule is said to "fire."

The absence of a condition can also be tested. This has the form:

```
¬P(x,y,z) -> ...
```

Here the premise would be true if there were no ternary tuples in relation P. The interpretation of the absence of a tuple as the negation of its presence is dependent upon the assumptions of the programmer. Absence and negation are not necessarily synonymous.

At this point, the binding of free variables should be discussed. Bindings of free variables remain in effect only for the duration of the rule. In other words, the scope of a free variable within a rule is confined to that rule. Free variables are not bound in a test for absence. Thus, the variables in the implication ¬P(x,y,z) -> ... remain unbound.

Constraints may also be used in a premise. Our example could be written:

```
if P(x,y,z), x < 8 -> ...
```

where x < 8 is a constraint. Constraints may be any boolean expression.

The second part of the rule is the conclusion. Conclusion segments of an implication may be used to alter the state of the system. Consider the following:

```
if P(x,y,z) -> R(x,y)
```

If through pattern-matching, the rule fires (the premise becomes true), an assertion, **R(x,y)**, is established where the tuple **<x,y>** is added to the relation R. Remember the bindings of **x** and y in relation R will be the same as the bindings of **x** and y in relation P.

The use of deletion in the conclusion segment is shown as:

```
if P(x,y,z) -> R(x,y), ¬P(x,y,z)
```

This will result in the removal of the tuple (that became bound to **x**, y, and **z**) from relation P. This is quite common in Omega rules. If one or more conditions in the premise are not removed in the conclusion, the conditions that precipitated the firing of the rule would remain in effect. Thus, the rule would keep on firing. An abbreviated syntax can be used to denote the cancel operation:

```
if *P(x,y,z) -> R(x,y)
```

where ***P(x,y,z)** represents **P(x,y,z)** in the premise and **¬P(x,y,z)** in the conclusion.

One other syntactical extension is the use of **else** before an implication to establish a compound rule. Suppose we had the following:

```
if *P(x,y), *Q(m,n) -> R(m).
if *P(x,y) -> R(x).
```

In this example, we want the second rule to fire only when the first one fails. The first rule may never fire, however, since the matching of the tuple **<x,y>** in the P relation will fire the second rule. In this case, the example could be written as:

```
if *P(x,y), *Q(m,n) -> R(m)
else if *P(x,y) -> R(x).
```

The implication associated with the else statement will only be evaluated if the first implication fails.

A summary of the rule features is presented through the following example:

```
if *P(x,y),Q(m,4),¬R(s),x > 4 -> ¬Q(m,4),T(x)
else if *P(x,y) -> T(y).
```

It should be noted that although many rules may have their premises satisfied (the rules are "triggered"), only one rule is executed (fired) at a time. The indivisibility of rules can be used to support mutual exclusion of processes.

E. FUNCTIONS AND OTHER APPLICATIVE FEATURES

Named functions may be used to calculate components in a tuple. A function invocation is used in the conclusion of the following rule:

```
if *P(x,y,z) -> Q(rest[x]).
```

The argument to relation Q is a function (note the use of brackets for function calls) which returns a pointer to the tail of a list. In this example, variable x must be bound to a list in the premise condition of the rule. Function invocations may also be used as constraints in a premise:

```
if *P(x,y,z), first[x] < 10 -> ...
```

In this case, variable x must be bound to a list which has a first element less than ten.

New functions are declared as follows:

```
fn number_items [list]: if list = Nil -> 0
          else 1 + number_items[rest[list]].
```

The body of a function is a conditional expression similar in form to a rule. There are two qualifications. The premise can only be a boolean expression, while the conclusion can only be another expression. This ensures the absence of side effects. Note that the conclusion contains a recursive call. There are no iterative constructs in Omega.

Functional bodies clearly display six desirable characteristics that can be obtained through the use of expressions. These characteristics include "transparency of meaning and purpose, independence of parts, recursive application, narrow interfaces, and manifestness of structure [Ref. 17:: p. 16]."

Applicative expressions can also be used to calculate the value of an argument in a tuple. Consider the following:

```
if *P(x,y,z), y + 2 > 10, z - 1 < 5 ->
                Q(x,5 + z).
```

In this example, infix operators are used to calculate two constraints in the premise. One infix operator is also used to calculate an argument in the assertion of the tuple <x,5+z> for relation Q. All variables must be bound prior to participating in an applicative expression.

F. PROCEDURES

Procedure calls are quite similar to the function invocations discussed in the previous section. Both processes return results which may be used in expressions. The underlying mechanism of a procedure call is quite different, however, from that of a function. One important difference is that side effects are possible in procedure calls. This is a result of the use of rules to implement the actions of a call.

A procedure call involves synchronous communication; that is, the sender (object or process) that made the assertion expects a reply before continuing. The sender is usually expecting one or more rules to be processed prior to receiving a response. Procedure calls are distinguished from conventional relations by enclosing the asserted tuple in braces. This is illustrated in the following example:

```
if *input(x), state(q) -> push{x,mystack}.
```

The relation push is a procedure call. It will be translated by the system into the assertion push(a,x,mystack). The object a is a system-supported relation that represents the sender. The relation a will be used as a mailbox to hold the response from an active rule that contains the push relation. This rule could be written as:

```
if *push(a,x,stack), *contents(list,stack) ->
                    a(stack),
                    contents(cons[x,list],stack);
```

By convention, a is placed as the leftmost member of the tuple <a,x,stack>. With the assertion of a(stack), the sender may obtain the result and continue with the computation. The tuple <a,x,stack> could be compared to the establishment of a conventional activation record, while the mailbox a is analogous to the activation record of the caller.

The above example shows that the result of a procedure call does not have to be used in an expression. In this case, the returned value is used only for synchronization. Another procedure call which does not use the return value is the system-defined display procedure. This procedure sends a message to the screen. An example of a call that uses the return value would be:

```
if *P(x,y,stack) ->
```

```
Contents(cons[pop{stack},x],y).
```

In this case, the result from popping the stack will be appended to a list that is bound to variable x.

G. SEQUENTIAL CONTRCI

Consider the following rule from a solution to the Towers of Hanoi problem:

```
if *move(user,1,source_peg,destination_peg,
                          auxiliary_peg)
->
   display {"Move disk 1 from peg "},
   display {source_peg},
   display {" to peg "},
   displayn {destination_peg},
   user(Nil);
```

Naturally the programmer would like the message to print in order. This will not necessarily occur, however, with the current structure of the rule. No order is assumed for evaluating the conditions of the premise, and no order is assumed for executing statements in the conclusicn. Further, there is no crder associated with the evaluation of multiple production rules.

A mechanism is therefore needed to give the programmer contrcl over processes which transition through multiple states. The solutien is a pair of braces. An open brace depicts the beginning of a sequential block, and a closed brace terminates the sequential block. The previous rules could be rewritten as:

```
if *move(user,1,source_peg,destination_peg,
                          auxiliary_peg
->
  {
```

```
        display {"Move disk 1 from peg "};
        display {source_peg};
        display {" to peg "};
        displayn {destination_peg};
        user(NIl)
      }.
```

Variables bound in the premise will keep their bindings throughout the sequential block. Bindings will also be retained in nested blocks.

H. DIRECTORIES

It has been previously mentioned that relations and objects must be defined prior to their use. These definition statements are used to bind a global name to the desired object or relation. Global names are controlled through directories.

The original schema for Omega [Ref. 4: pp. 34-35] discussed the use of one public and a number of private directories. An obvious use for these directories is to enforce information hiding. A private directory can be used for access control as follows:

```
Define {Private,"Push",Newrel{}}.
```

The define procedure call makes an entry into a directory partition. The Newrel procedure call returns a new relation object which is a unique identifier. The name push is bound to the new relation object in the private directory.

The creation of a new relation associates full access rights (capabilities) with the relation name. The access rights are read, add, and delete. It is possible to restrict access rights:

```
Define {Public,"Push",AddOnly{Push}}.
```

The **AddOnly** call creates a copy of the system identifier that has been bound to the private name **Push**. The copy differs in that its rights have been restricted. The new identifier is then placed in the public directory where it can be generally accessed in accordance with its restricted access rights.

Public and private directories were not defined in the McArthur prototype. A single directory named **root** was implemented as shown in the following definition.

```
Define {root,"Push",newrel{}}.
```

I. PRODUCTION RULE SYSTEM

The previous production rules are examples of active rules in the Omega system. These rules are normally entered into a file using a standard text editor (they can also be entered interactively). Active production rules constantly monitor the relations in their premises on a test-fire basis. A rule denotation (<< >>) is used to syntactically distinguish the production rules from command rules (section J). A possible rule definition is:

```
Define {root, "SampleRules,
  <<
    if *top_item(a,stack), contents(list,stack)
    -> a(first[list])
  >>}.
```

The rules have been entered in a passive status, basically parsed but not evaluated. To make the rules active, the procedure **Act** is used:

```
Act {SampleRules}.
```

The rules are then elevated to an active test-fire status. It is possible under the original Omega design to activate

and deactivate rules throughout a program. Rule deactivation was not implemented, however, in the McArthur interpreter.

J. INTERACTING WITH OMEGA

A second category of rules is the command rules. These rules are used to interact with the system. Unlike production rules (which when activated are constantly evaluated), the command rules are only evaluated once. The evaluation sequence for command rules is test, fire, and forget [Ref. 14: p. 30].

One useful application of command rules is queries. An example of a session with Omega that combines the command rules with the production rules is presented below:

```
Define {root,"Married",newrel{}}.
Define {root,"Brother",newrel{}}.
Define {root,"Bill",newrel{}}.
Define {root,"Karen",newobj{}}.
Define {root,"Joe",newobj{}}.
Define {root,"Jane",newobj{}}.

Define {root,"Samplerules",
  <<
    if *Brother(x,Joe) -> Married(x,Karen);
    if *Brother(x,Bill) -> Married(x,Jane);
  >>}.

Act {Samplerules}.
```

The definitions and production rules could have been entered interactively or from a file. If a file is used, the file must be activated with the procedure Do.

Suppose the user wanted to establish that Bill was the brother of Joe. He would enter Brother(Bill,Joe). The

first production rule would fire thus asserting that Bill is married to Karen (**Married(Bill,Karen)**). Next the user might enter:

```
if Married(Bill,Karen) -> Display{"Yes"}.
```

Since **Married(Bill,Karen)** has been asserted, Omega will respond with **Yes.**

III. OMEGA-1.5: DESIGN ISSUES

A. GOALS

Four different syntactical forms have been suggested for the Omega language [Ref. 15]. The second and third alternative forms suggest a pseudo-natural style that provides a greater degree of readability for novice (and experienced) users of the language. Readability has long been an issue in the development of computer languages. A 1973 memorandum by C. Hoare listed readability as one of the top five objective criteria for good language design [Ref. 17: p. 6]. The goal of the Omega-1.5 grammar was to develop an independent design that fulfilled the intent of the Omega-2 and Omega-3 grammars (primarily Omega-2) and to test the feasibility of implementing such a design. Inherent in this objective were the following design characteristics:

- Readability. The 1.5 grammar had to offer a notable increase in readability over the predicate logic style notation of Omega-1.
- Simplicity.
    1. The engineering solution must be simple and practical.
    2. The syntax should have a close correlation to the Omega-1 syntax. The original thought was to have an injective mapping (after the removal of the noise words) from Omega-1.5 to Omega-1 (a function f:A->B is injective if a<>a' implies f(a)<>f(a')).
    3. There should also be a close correlation between a translated version of Omega-1.5 program and a program written in Omega-1.

4. Additional definition statements should be kept to a minimum.

- Flexibility. A programmer should have the capacity to write a relation in many ways, depending upon the context. The design should also be extensible. A programmer should be able to augment the given collection of noise words as necessary.

Upon completion of the design, a translator was built to test the design's feasibility. Sample programs were then written to evaluate the completed design against the initial design goals.

B. REPRESENTING OBJECTS AND VARIABLES

Let us review the last example written in Omega-1:

```
Define {root,"Married",newrel{}}.
Define {root,"Brother",newrel{}}.
Define {root,"Bill",newobj{}}.
Define {root,"Karen",newobj{}}.
Define {root,"Joe",newobj{}}.
Define {root,"Jane",newobj{}}.

Define {root,"SampleRules",
<<
  if *Brother(x,Joe) -> Married(x,Karen);
  if *Brother(x,Bill) -> Married(x,Jane);
>>}.

Act {SampleRules}.
```

Bill, Karen, Joe, Jane have been defined as objects in this short program. The variable x represents an unbound variable. The following code is the same program written in Omega-1.5:

```
"Married" (procedure) is defined as a relation.
"Brother" (procedure) is defined as a relation.
"Bill" (procedure) is defined as an object.
"Karen" (procedure) is defined as an object.
"Joe" (procedure) is defined as an object.
"Jane" (procedure) is defined as an object.

"Sample_rules" (procedure) are defined as
  Rules
    If given a person is the brother of Joe
      then the person is married to Karen;
    If given a person is the brother of Bill
      the the person is married to Jane;
  end_rules.

The sample_rules (procedure) are activated.
```

Objects and variables may be written in the 1.5 grammar as:

```
word_phrase = noise_preps? noise_word? word
```

The question mark means optional. A word is simply an identifier, as classified by a scanner. The noise word category includes the indefinite articles a and an and also the definite article the. Noise_preps is an extensible category (chapter IV) that represents prepositions. Noise_preps is defined:

```
noise_preps = noise_prep
            | noise_prep noise_preps
```

The symbol "|" means or. A noise_prep is simply a preposition. Notice the optional recursive call in the definition of noise_preps. This permits the use of multiple word prepositions. Examples of common prepositions are:

to

with

for
in addition to
according to

Legal instances of the word_phrase category in Omega-1.5 include:

person
the person
with the person
with George
to George
according to George
according to the person

Variables and objects can therefore be used as subjects, as direct objects, and as indirect objects. With the removal of the prepositions and articles, we have a bijective mapping for objects and variables in the Omega-1.5 and Omega-1 grammars. Thus, the translation of objects and variables is quite simple.

C. RELATIONS

1. Names

In Omega-1, a relation is named by an identifier that becomes globally bound through a definition statement. The name is then used in association with asserted tuples of that relation as shown here for the relation contents:

```
contents(1,x,y)
```

In Omega-1.5, an identifier is also defined and globally bound as a name for a relation. The use of the identifier is directed, however, by the following rule:

```
relation_phrase = noise_verb not? noise_verb?
                                  word_phrase
```

The word_phrase category was defined in the previous section. Clearly, it has multiple uses. The noise_verb category represents copulative and auxiliary verbs. One noise verb is required, although two are possible.

One use of the relation_phrase category is as a verb phrase. This combines an auxiliary verb with a main verb. Examples of auxiliary verbs include is, do, has, and are. The main verb would be the defined identifier, as previously described. Valid examples of the relation_phrase category might be:

can write
should study
are playing

Rules can be written in multiple tenses, depending upon the proper selection of verb tense and auxiliary verb. The present tense can be achieved by using the emphatic word do as the auxiliary verb:

I do study

The addition of the optional second noise verb in the relation phrase definition permits the use of most combinations of the active, passive, and progressive active voices for the six basic tenses in English. Table I gives several examples. Tenses such as the future perfect combined with the progressive active voices are not permitted as they require three auxiliary verbs:

I will have been calling

Verb phrases can be easily negated. Recall that in Omega-1, a test for an absence of a tuple is written as:

¬contents(1,x,y)

TABLE I

Verb Tense Examples

EXAMPLE	VOICE	TENSE
I had called	active	past perfect
I was calling	progressive active	past
I had been called	progressive active	past perfect
I will be called	passive	future
I will have called	active	future perfect

This is accomplished in Omega-1.5 by using the optional word **not** after the first auxiliary verb:

are not playing
do not study
have not been called

Applications of the relation_phrase category can also be written using copulative verbs. Copulative verbs are verbs which connect a subject with its complement. The complement is either a predicate noun or a predicate adjective. The defined identifier becomes the complement when a copulative verb is used. Examples of a copulative verb with a predicate noun as its complement are:

is the contents
is a member
are the components

Examples of predicate adjectives with copulative verbs are:

is ill
are happy

felt sick

As with verb phrases using auxiliary verbs, the verb phrases with copulative verbs are also easy to use in a check for an absence of a tuple. The optional word **not** is placed after the copulative verb:

is not the contents
is not a member
is not ill

2. General

In Omega-1, a relation was viewed as an object name combined with a tuple. Sample relations could be:

```
push(user,item,stack)
contents(list,stack)
know(I,proposition)
```

These relations might be written in Omega-1.5 as:

```
A user does push an item on a stack
A list is the contents of the stack
I do know the proposition
```

Figure 3.1 shows the mapping between relations in the two grammars.

In Omega-1.5, the first argument to a tuple is placed before the relation name. This becomes the subject. The predicate consists of the relation phrase and the rest of the arguments in the tuple. These remaining arguments are essentially indirect and direct objects. The grammar specification is:

```
subject relation_phrase arguments
```

where the subject is an expression and the arguments category calls for zero or more expressions. If in the

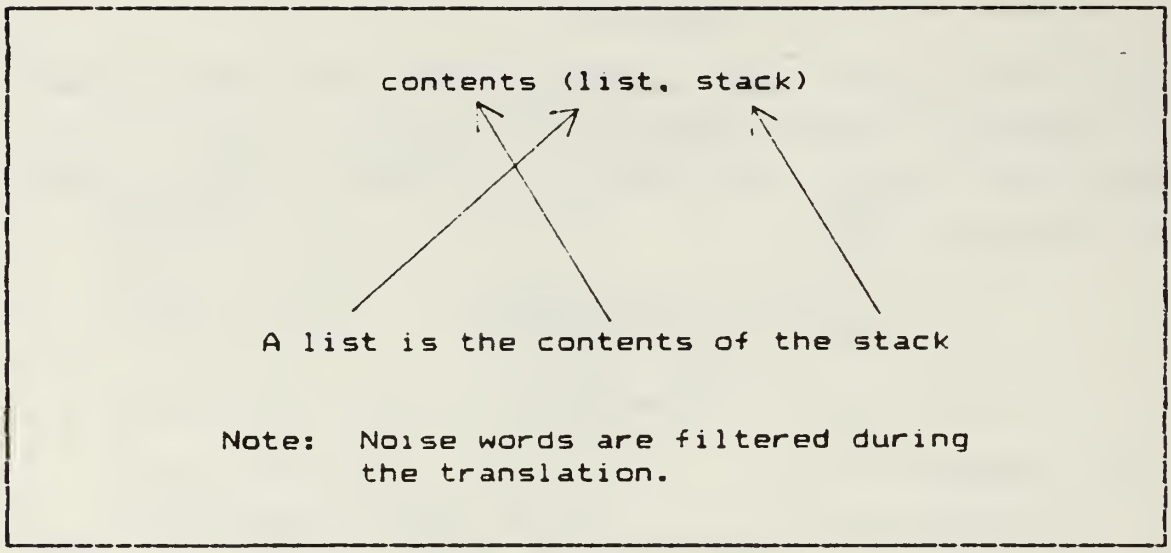

Figure 3.1 Mapping Between Relations.

arguments there is more than one expression, a noise preposition must be inserted between each of the expressions. The rationale for the preposition requirement is discussed in the implementation chapter (Chapter 4). This restriction is not a great burden. If a programmer wants to write:

give(man, dog, bone)

in Omega-1.5, he would want to write:

A man did give a dog a bone.

The preposition requirement would necessitate the relation to be written as:

give(man,bone,dog).

and translated as:

A man did give a bone to a dog.

D. LISTS

Lists have previously been described as one of the three value components in the system. A sample list expression in Omega-1 is:

[red,white,blue]

This would be written in Omega-1.5 as:

the_list of red, white, blue
or the_list of red, white, and blue

The start of a list is signaled by the term the_list. This again is an extensible category and will be discussed in chapter four. The_list maps to the open and closed brackets in the Omega-1 syntax (see Figure 3.2).

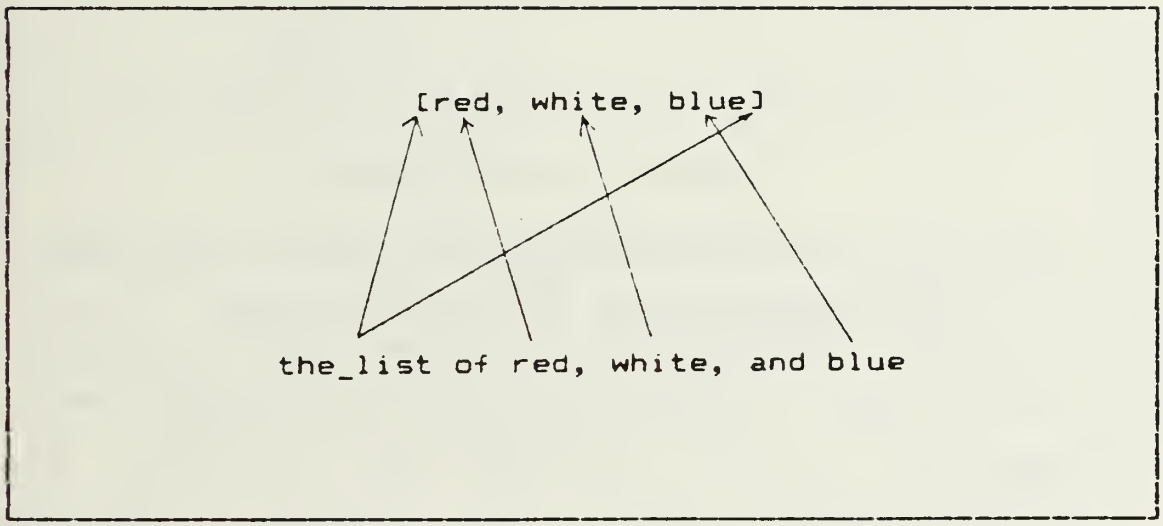

Figure 3.2 Mapping Between Lists.

The comma between arguments of a list written in Omega-1.5 is optional. Consequently, the list

[man,implies,mortal]

could be written as:

the proposition man implies mortal

if **proposition** had been defined as an extension to the list_starter category. The omission of commas created several implementation problems, and its use is therefore questionable.

E. FUNCTIONS

1. Invocation

There are two types of function calls in Omega-1.5. One type is similar to the relation format described in section C of this chapter. The second type differs in not inverting the function name with the first argument. Both types are described in the following definition:

```
fn_application = word_phrase '(' 'function' ')'
                                        arguments
               | subject '(' 'predicate' ')'
                 relation_phrase arguments
```

The first alternative is the case without the inversion of the first argument with the function name. In the original design, this was the only format for a function call. Common Omega-1 functions that fit this category are:

```
first[list]
cons[item,list]
rest[list]
```

The Omega-1.5 form would be:

```
the first (function) of a list
the appending (function) of an item with a list
the rest (function) of a list
```

The notation **(function)** is used to signal the function call. It maps into the left and right brackets (see Figure 3.3). It can be used in a program to readily spot function calls without slowing down their comprehension. Specific words were considered to represent the left bracket in lieu of the term **(function)**, but it was felt that a specific word would limit the possibilities for expressing function invocations in a naturalized style.

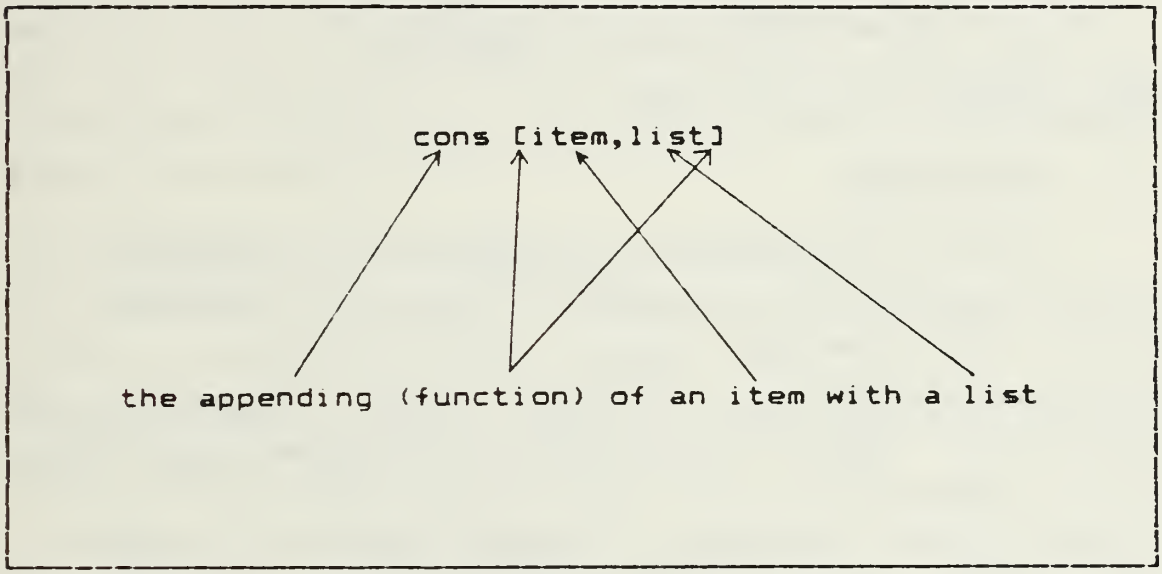

Figure 3.3 Mapping Between Functions.

During the implementation phase, it became obvious that having only one format for function calls was not sufficient. Consider the following built-in functions in the McArthur interpreter:

```
IsList[item]
IsInt[item]
IsStr[item]
```

It would be extremely difficult to write these functions in Omega-1.5 without inverting the argument and the function

name. Thus the second part of the fn_application definition was added. This format basically applies to any function that calls for a true/false or yes/no condition (the function does not necessarily have to return a true/false or yes/no). The rule:

```
If *IsStr[item] -> display{"true'}
```

can now be written:

```
If given an item (predicate) is a_string
  then "true" (procedure) is displayed
```

where a_string maps into IsStr.

2. Declaration

Function declarations in Omega-1.5 are similar to function declarations in Omega-1. The only difference is that the word **function** appears in its entirety in the heading instead of being abbreviated as **fn**. The decision to keep the same heading was based on the mathematical nature usually associated with this applicative component. A function call within a definition, however, (such as a recursive call) would be written in the naturalized style. An example of a function call in Omega-1 and its translation in Omega-1.5 is presented below:

```
Omega-1:  fn number_items[list]:  if list = Nil
                -> 0
              else 1 + number_items[rest[list]].

Omega-1.5: function number_items[list]:
             if list = Nil then 0
             else 1 + the number_items (function)
             in the rest (function) of the list.
```

F. PROCEDURES

Recall that in Omega-1, procedure calls were almost syntactically identical to the assertion of a relation. The only distinguishing item was the use of braces in the place of the parentheses:

relation: contents(5,list)
procedure call: pushed{5,stack}

Procedure calls in Omega-1.5 are also identical to Omega-1.5 relations with the exception of one distinguishing element. The symbol **(procedure)** is used to identify the requirement to surround the arguments with braces instead of parentheses during translation:

relation: 5 is the contents of the list
procedure call: 5 (procedure) is pushed on the stack

Nested procedure calls in Omega-1.5 would appear inside out from a similar structure in Omega-1:

Omega-1: a{b{c{d}}}
Omega-1.5: d (procedure) c (procedure) b (procedure) a

Figure 3.4 shows the mapping between procedure calls in the two grammars.

G. GENERAL

Appendix B shows the translation of the most common symbols that are not translated literally from Omega-1.5 into Omega-1. The word given is used to replace the cancellation symbol * in Omega-1. The rule markings, << and >>, are respectively represented as Rules and end_rules. Sequential control braces are replaced by begin and

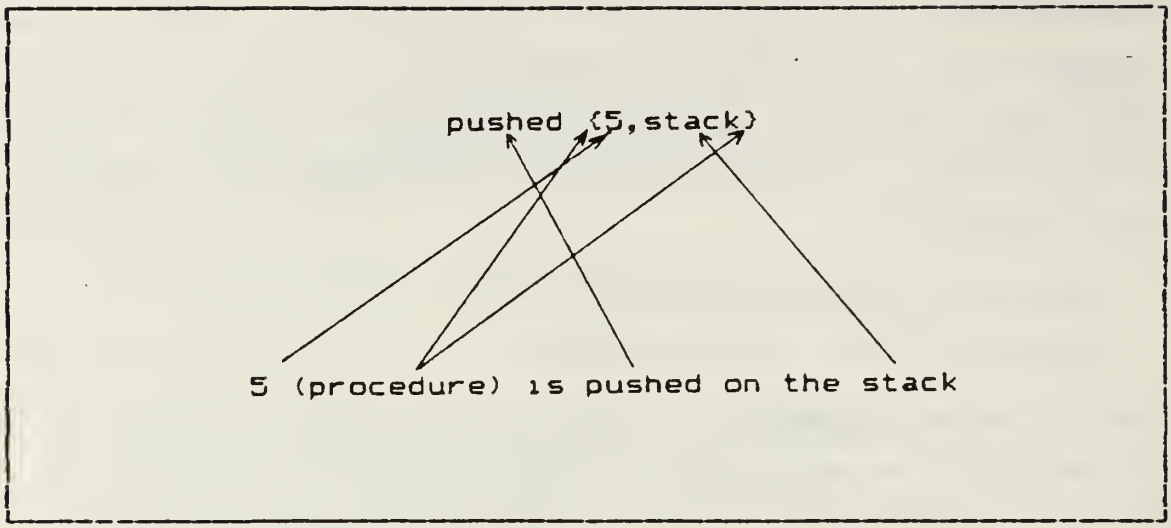

Figure 3.4 Mapping Between Procedures.

end_block. This is similar to the block control structures begin and end in conventional languages such as Pascal. Appendix C lists sample programs which illustrate syntactical structures in the Omega-1.5 and Omega-1 grammars.

IV. IMPLEMENTATION AND APPLICATIONS

A. GENERAL

A quick translator was developed to test the design decisions of the Omega-1.5 grammar. There were two goals in the development of the translator. The first goal was simply to evaluate the feasibility of the 1.5 grammar. A second goal was to explore extensible options to create a flexible environment for the grammar. The implementation language was Turbo Pascal [Ref. 18]. Pascal was chosen for its simplicity as a high-level language. Turbo Pascal was selected as one of the better Pascal environments for prototype programming. The built-in editor and Turbo's speed of compilation facilitated the testing and changing of various design decisions. The inefficiency in the object code and the lengthy run-time system that is added during compilation were not considered to be serious hindrances.

B. SCANNER AND PARSER

The stages of the translator were arranged in a pipeline configuration. The scanner processes input characters to recognize tokens. As a token is recognized, it is fed into the parser. Token classes consist of identifiers, delimiters, strings, and integers. Identifiers are described as:

```
(letter) (((letter) | (digit) | _)*
                      ((letter) | (digit)))*
```

where letters can either be upper or lower case. A digit is simply an element of the set (0..9).

The scanner has two filters. The first screens out comments, and the second filters characters that are not in

the 1.5 grammar (such as tabs). A carriage return and line feed is converted to a space.

Parsing is done in one pass by recursive descent. This required the Omega-1.5 grammar to be massaged into IL(1) form by removing left-recursion and nondeterminism.

C. TRANSLATION PROCESS

Translations are performed straight off the parse. Brief comments on some of the more intricate aspects of the translation process will be presented. Problem areas will be highlighted.

Previous figures have shown a mapping from several 1.5 statements into the Omega-1 syntax. These figures illustrate the switching of the first argument and the relation name. The relation:

A list is the contents of the stack

comes out of the pipeline as:

list (contents, stack)

In this case, contents and list must be switched (see Figure 4.1a). This was relatively easy.

Figure 4.1b shows a more difficult situation. Here, the first argument is a function call. The entire function call must be switched with the relation name. Additional complexity could result with nested calls. The solution was to have an output buffer consisting of an array of strings. The function call, first[list], was converted during the translation into a single string. Thus, the switch only involved two strings.

Another difficulty was inserting commas between arguments. A comma must be placed after each argument in a tuple with two exceptions. One exception is with a unary

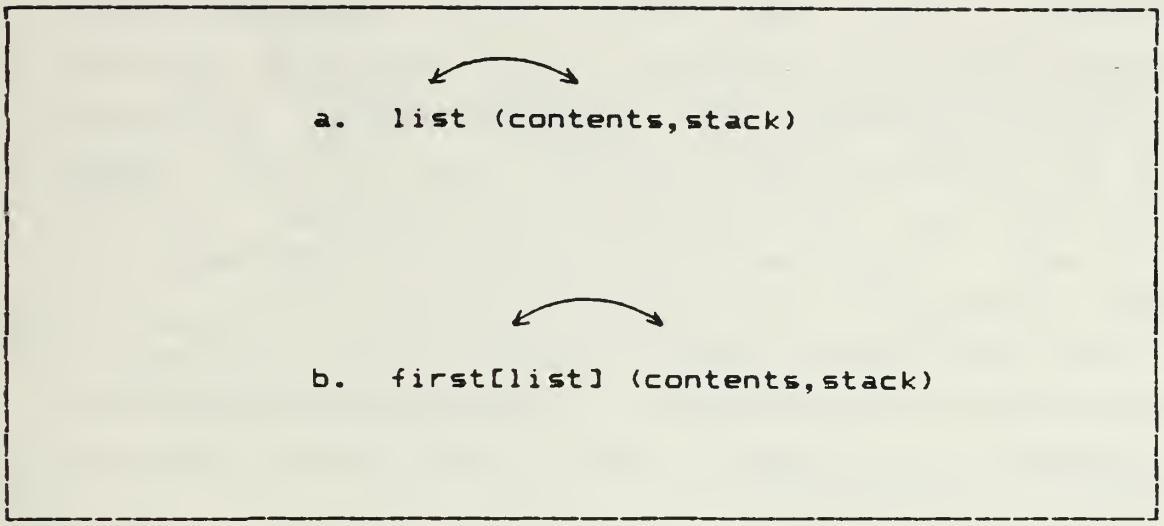

Figure 4.1 Switching Relation Name and First Argument.

tuple. The second exception is after the last argument in a multiple-argument tuple. Commas therefore could not be inserted without looking ahead in the translation.

The open parenthesis of a relation in Omega-1 does not have a corresponding component in Omega-1.5. In the implementation, the auxiliary verb without a (function), (predicate), or (procedure) in front was used to map into the open parenthesis. This was straightforward. The challenge was in closing the open parentheses, brackets, and braces. Flags had to be set to insert the proper closing after the last argument. This is compounded with:

The appending (function) of the item with the
list is the contents.

Here, a closed bracket must be inserted before a closed parenthesis:

contents(cons[item,list]).

Noise words (prepositions, articles, auxiliary verbs) are filtered during translation, as previously discussed.

Extensions to the ncise words (section D) are declared through a definition statement. These definition statements must alsc be filtered. In the implementation, the definition was translated into the buffer, with the buffer pcinter being reset upon discovering that the definition was for a noise word. Thus the definition was deleted before the buffer was dumped to the output file.

One final implementation problem involved the use of prepositions between arguments. Originally, this was not a requirement. A problem could result though with the following two asserticns:

A list (procedure) is pcpped off a stack.
A list (procedure) is pushed on a stack.

This would be translated as:

popped{list,stack}.
pushed{list,stack}.

If the period after the first statement was accidentally omitted, however, the error would be accepted with the following translation:

popped{list,stack,pushed{list,stack}}.

By inserting the mandatory preposition after the second argument, this condition is avoided. Lists that are written without the comma option between arguments would still face this problem.

D. EXTENSIONS

A key element in Omega-1.5 is the ability to add noise words. This feature is essential if flexibility in the design is to be achieved. The extensions primarily apply to auxiliary verbs, to prepositions, and to words which signal the start of a list.

1. Noise Verbs

Four auxiliary verbs were built into the grammar: is, are, has, and do. Suppose, however, that we had the relation:

```
if *pop(user,stack) -> ...
```

and we wanted to translate it as:

If given a user does pop a stack then ...

We would need to define the auxiliary verb does. This is done by:

"Does" (procedure) is defined as a noise_verb.

The list of noise verbs was implemented as an array of strings. The array ceiling was arbitrarily set at twenty. The definition statement is not translated--in this case it simply adds the word does to the list of noise verbs. When an auxiliary verb is expected during the parse, the translator does a sequential search through the array.

2. Noise Prepositions

Noise prepositions were handled in the same manner as noise verbs. Eleven common prepositions were built into the translator (and the grammar). One of the eleven built-in prepositions is actually a conjunction rather than a preposition, since programming experience showed that the conjunction and could add a great deal of clarity in many situations that called for an optional preposition. The following example shows a suitable case. In this instance, and is used to connect two indirect objects:

```
Omega-1:  donate(man,money,poor,needy)
Omega-1.5:  A man did donate money to the poor
                                and to the needy
```

Without the use of and, the statement would read:

The man did donate money to the poor along
with the needy

This is quite awkward.

The large number of built-in prepositions was originally meant to cover the majority of situations in which a preposition would be needed. This violates the zero-one-infinity principle [Ref. 2], however, and a lesser number might be more effective. Noise prepositions are indicated in definitions by the term noise_prep:

"Off" (procedure) is defined as a noise_prep.

This definition would be required before the following procedure call would be made:

The top_item (procedure) is popped off the stack.

3. Noise "List-starters"

The third area of extensions involves lists. Recall that the start of a list was indicated by the default term the_list. Therefore the list [red,white,blue] could be written as:

the_list of red, white, and blue

In a previous example, a relation was listed as:

```
perform(compilers,[scanning,parsing,
                   code_generation]).
```

This relation could be written as:

Compilers do perform the operations of scanning,
parsing, and code_generation.

Operations would map into the left bracket, thus signaling the start of the list. First, however, operations would have tc be appropriately defined:

"Operations" (prccedure) is defined as a
list_starter.

E. ADDITIONAL CONSIDERATIONS

Two late changes were made to the translator after evaluating sample test prcgrams. In the initial design, and was used instead of a comma to connect multiple inquiries and multiple assertions. The translator keyed off the word and to determine the end cf the arguments in a tuple. And thus had a reserved word status, and could not be used as a noise preposition. This has been shown to be a severe limitatien. Omega-1.5 was amended to require the use of a comma to connect multiple inquiries. The conjunction and was inserted into the grammar as an option after the ccnnecting comma. If and is present, the translator simply discards it.

A final change dealt with definition statements. The McArthur prototype implemented only one directory named root, which appears as the first argument in a definition statement. As the first argument, root would therefore become the subject in an Omega-1.5 definition. Programming experience demonstrated that the second argument wculd make a more logical subject. Consequently, the one directory is cmitted in Omega-1.5 definitions. An Omega-1.5 definition:

"Contents" (procedure) is defined as a relation.

is translated as:

defined{"contents",newrel{}}.

This would have to be converted to:

```
define{root,"contents",newrel{}}.
```

in order to run on the McArthur prototype. A conversion rule was then added to the McArthur interpreter. Whenever the interpreter is invoked, the following rule becomes active:

```
define{root,"defined",newrel{}}.
define{root,"DefMap",
<<
  if *defined(a,n,x) -> define{root,n,x},a(n)
>>}.
act{DefMap}.
```

Rules of the form:

```
defined{x,newrel{}}.
```

are now automatically converted to:

```
define{root,x,newrel{}}.
```

When multiple directories are implemented, it is envisioned that the directory name will be used as an adjective before the term **relation**:

```
"Contents" (procedure) is defined as a private
                                        relation.
```

This is easily implemented with the addition of the following rule into the previous rule definition:

```
if *defined(a,n,t,x) -> define{t,n,x}, a(n)
```

F. SAMPLE APPLICATIONS

Five application programs were written to test the design and implementation of the Omega-1.5 grammar. The Omega-1.5 programs and their Omega-1 translations appear in Appendix C. The following discussion provides a brief explanation of each program.

1. PDA

PDA is a program that simulates a deterministic pushdown automaton that accepts strings in:

$$\{wcw^R \mid w \text{ in } (0 + 1)^*\}$$

A pushdown automaton was selected because it requires a stack, and thus would be an excellent example for illustrating a stack abstract data type in Omega-1.5.

Abstract data types are very easy to program in Omega. The naturalized format of Omega-1.5 adds a significant degree of clarity and readability to the abstract data type rules. In a sense, the rules are almost self-documenting code.

2. Logic5

Logic5 is a program which demonstrates the use of simple logic in Omega. For example, if the database contains the following concepts:

Fido is a dog.
Dog implies animal.
Animal implies mortal.

and a query is made as to whether Fido is mortal, the program can properly conclude an affirmative response. The first example of Logic5 (Appendix C) was written before the Omega-1.5 study. It was selected because of its reliance on

the use of lists. Lists can be difficult structures to translate into a naturalized language. The other two Logic5 programs in Appendix C are the Omega-1.5 version and the resulting Omega-1 translation. The Omega-1.5 version relies upon the list_starter extension option. This version shows the added readability obtained from using the list_starter extension.

3. Towers of Hanoi

The Towers of Hanoi program can be simply solved using three rules. This is shown in the first Towers of Hanoi example in Appendix C. This example was taken from [Ref. 14]. Note the heavy reliance on recursion. The second Towers of Hanoi example shows the Omega-1.5 version. The Omega-1.5 program introduces greater readability, but at a high cost. It was difficult to verbally describe the semantics of the program. Perhaps the predicate logic style might be more advantageous for short programs with extensive recursion and numerous applicative components.

4. Zoo

The Zoo program was derived from [Ref. 19]. It is a prime example for illustrating the use of a naturalized style for rule-based systems. The Zoo program is a toy analysis system which identifies animals. It involves a robot (Robbie) visiting a zoo. Robbie would like to be able to identify the various animals. He can see elementary features such as size and color, but he cannot combine the facts to form conclusions like "this is a zebra."

To make the reasoning procedure more stimulating, intermediate facts are generated. Thus Zoo produces chains of conclusions which lead to the identification of a particular animal. To limit the number of required rules, we suppose that the particular section in which Robbie is

visiting contains only seven animals. The rules can be understood by examining how Robbie would try to analyze an unknown animal.

The observed animal has a tawny color and dark spots. Rules 9 and 11 are suggested. Neither is triggered, however, since both have additional antecedent conditions to be met. Robbie notices that while nursing a baby, the animal chews its cud. Evidently the animal gives milk. This fact fires rule 2, establishing that the animal is a mammal. Since the animal is a mammal and the animal chews its cud, rule 8 fires. Thus it is established that the animal is an ungulate, and it has two or four toes per foot. Next Robbie notices that the animal has long legs and a long neck. Rule 11 fires, and the animal has been identified as a giraffe. The process is modeled by Figure 4.2.

Table II shows a comparison between the rules defined in [Ref. 19], the Omega-1.5 version, and the Omega-1 notation. Note the close similarity between the original rules and the Omega-1.5 rules. A forward-chaining, deduction-oriented, rule-based system appears to be ideal for Omega, especially for the 1.5 syntax.

5. PI-1

The final example shows a more serious application. It includes the rules and definitions for an interpreter/unparser for a simple language of arithmetic expressions composed of +, -, x, /, parentheses, and literal integers. Syntax-directed editing rules are also provided. The Omega-1 version was developed by Bruce MacLennan and discussed in [Ref. 20]. It was a relatively easy task to write the Omega-1.5 version. The added clarity of the Omega-1.5 syntax is quite obvious in this example.

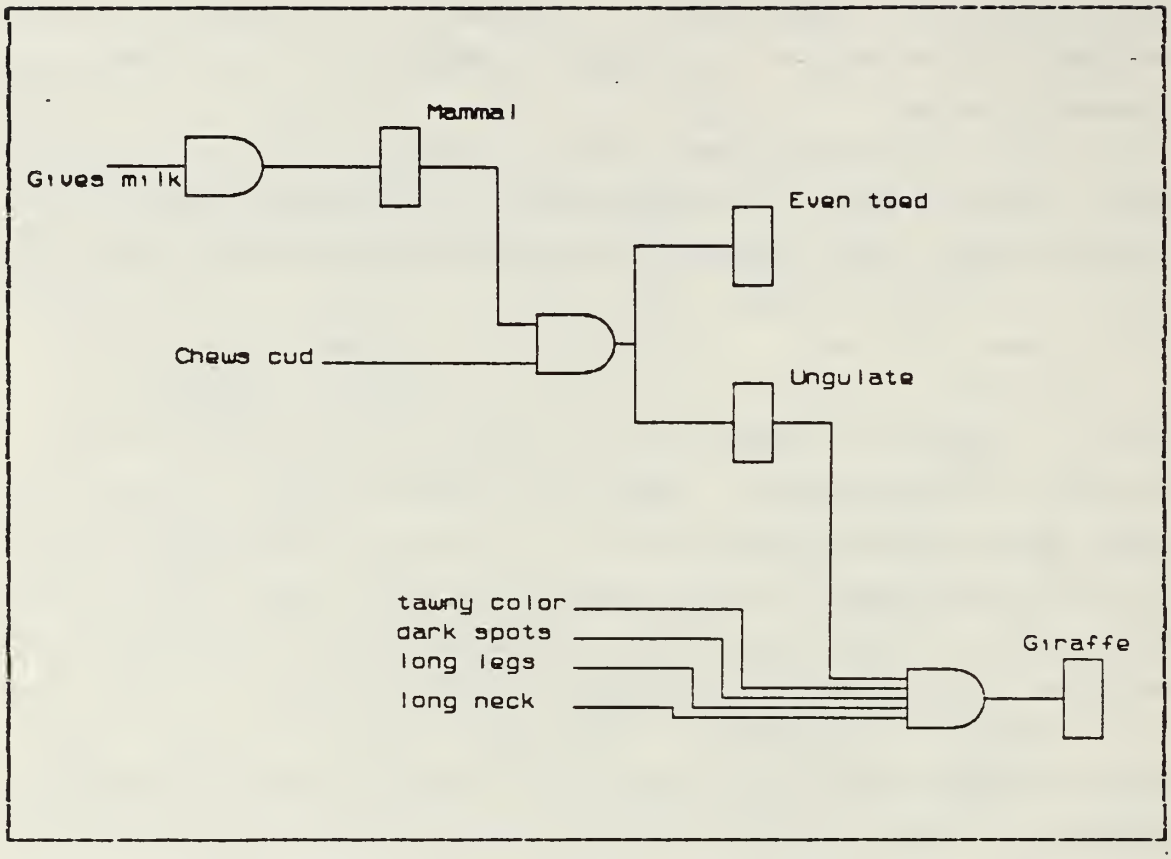

Figure 4.2 An Example of Forward-Chaining.

G. PROGRAMMING CONSIDERATIONS

Many useful insights were gained from programming the sample application programs. These insights are provided to assist future Omega programming efforts with the naturalized syntax.

If the relationship name is to be a verb (as opposed to a predicate noun or a predicate adjective), use the past participle form as much as possible (the past participle form of a regular verb is formed by adding ed to the infinitive). This form is quite flexible. All of the tenses in the passive voice can be written using an auxiliary verb and the past participle. Many tenses for the active voice, such as the present perfect, also use the past participle. Thus

TABLE II

Rule Comparison

Defined Rules:

```
If the animal is an ungulate
   it has long legs
   it has a long neck
   it has a tawny color
   it has dark spots
then it is a giraffe

If the animal is a bird
   it does not fly
   it has long legs
   it has a long neck
   it is black and white
then it is an ostrich
```

Omega-1.5:

```
If given the animal is an ungulate,
   the animal has long_legs,
   the animal has a long_neck,
   the animal has a tawny_color, and
   the animal has dark_spots
then the animal is a giraffe

If given the animal is a bird,
   the animal does not fly,
   the animal has long_legs,
   the animal has a long_neck, and
   the animal is black_and_white
then the animal is an ostrich
```

Omega-1:

```
If *ungulate(animal),
   long_legs(animal),
   long_neck(animal),
   tawny_color(animal),
   dark_spots(animal)
-> giraffe(animal)

If *bird(animal),
   ¬fly(animal),
   long_legs(animal),
   long_neck(animal),
   black_and_white(animal)
-> ostrich(animal)
```

the past participle can be used to show action in the past, present, and future:

If the item has not been pushed then the item will be pushed.

(passive present perfect) (passive future)

In some instances, the defined relation name is of a form that is just not suitable for that particular instance. An ideal solution would be to have an alternate synonym that is recognized as the same relation. This is possible in Omega. For example, in the Logic5 program, the relation ask was defined in its present part, as the relation was going to be predominantly used in the future tense. Several antecedent conditions required the present perfect tense, however, and thus the past participle was needed instead of the present part. The solution was to establish the following definition:

"Asked" (procedure) is defined for ask.

Consequently, both asked and ask could be used to identify the same relation.

Although the synonym definition is possible, it should be used only when other options are not feasible. Excessive synonyms clutter the program with extra definitions.

Since Omega-1.5 does not make a distinction between upper and lower case words, be careful to distinguish between relation names and unbound variable names. For instance in Omega-1, Top could be a relation, and top could be a variable. In Omega-1.5, the variable top would have to be written in a different form, such as top_item.

A final comment on Omega-1.5 programming is that an extensible noise word should be (as much as possible) the same part of speech as the category for which it is being defined. For example, the word questionable could be defined as a noise_verb. A rule could then be written:

If this is questionable programming then ...

and translated as:

programming(this) -> ...

This practice was found to add needless definitions to programs with little gain in semantic power.

H. DIFFERENCES FROM THE OMEGA-1 FOUNDATION

The sample applications have demonstrated four differences between the Omega-1.5 grammar and the Omega-1 form as implemented in the McArthur interpreter. One difference was mentioned in the previous section. The Omega-1.5 implementation does not recognize the distinction between upper and lower case in naming structures. Thus,

item
Item
itEm
IteM

will all be recognized as the same object. This convention was adopted since an object may appear as the first argument in an asserted relation and should therefore be capitalized as the first word in the sentence. Elsewhere, the lower case form would most likely be preferred.

Comments were handled differently in the Omega-1.5 implementation. Like Omega-1, Omega-1.5 signals the start of a comment by an exclamation point. Omega-1.5 differs though in that it also requires an exclamation point to end the comment. This permits in-line comments. It proved to be quite useful for such functions as numbering the rules in the Zoo example and adding clarity to the following rule in the PI-1 example:

```
If "abort" is the command, and
  given an expression is being evaluated
then !do nothing! .
```

In the Omega-1 implementation, statements could be terminated with either a period or a semicolon. The semicolon form means parse but don't evaluate. In the Omega-1.5 grammar, all statements must end with a period except for rules within a rule definition. As in Omega-1, rules within a rule definition must end with a semicolon. This convention was adopted to maintain a sentence-like appearance for assertions and definitions.

One final difference between the two implementations is that Omega-1.5 does not permit the establishment of a null tuple in a procedure or a relation. A subject is mandatory for each phrase. This limitation did not arise in the examples. The only two null tuple procedure calls that were found in Omega-1 were the **newrel{}** and **newobj{}**. These were built into Omega-1.5 simply as **relation** and **object** where **relation** maps to **newrel{}** and **object** maps to **newobj{}**.

V. OBSERVATIONS AND CONCLUSIONS

A. OBSERVATIONS ON OMEGA-1.5

1. Omega-1.5 Versus a Template Approach

At the beginning of the second chapter, it was stated that one of the goals of the Omega-1.5 grammar was to fulfill the intent of the Omega-2 template approach [Ref. 15]. The sample application programs have provided valuable experience which can be used to compare the Omega-1.5 grammar with a template format.

One key feature of Omega-1.5 is the flexibility that is achieved without adding numerous definition statements. Past, present, and future actions can easily be written. For example, in the PI-1 program, the following rule was used to translate the eval relation:

```
If given an expression is being evaluated,
  ...
then node1 must be evaluated, and
  node2 must be evaluated;
```

This could not be done with the template approach without adding a new template synonym each time a different tense is desired. Also with Omega-1.5 (as with Omega-1), the relation can be defined without first having to determine the context in which it will be used. This is not true for templates.

Another advantage of the Omega-1.5 design is the ability to handle tuples of various lengths in a relation. This feature makes procedure calls in Omega-1.5 (adding the mailbox) quite simple. The template approach would require a new template for each instance of a relation in which the

number of arguments did not agree with the original definition.

The negation of an inquiry or an assertion is also very easy in Omega-1.5. The word not is simply placed after the required auxiliary verb or copulative verb:

are not playing
is not a member

Negation in the template approach is formed by placing the word **not** after the first word of the template. This would be unsatisfactory for the following tuples from [Ref. 20: p. 9]:

_ pops _
_ pushes _ on _
_ receives _

One last strength of the Omega-1.5 grammar is that the Omega-1 translation is quite readable for individuals familiar with the predicate logic style. This is due to the direct mapping between Omega-1.5 and Omega-1. Relations in the template approach either would have to be translated into Omega-1 by assigning a number (R1 for example) rather than a name with semantic connotations, or by some other regular mechanism such as concatenating the template's parts:

template: _ pushes _ on _
translation: pushes_on (x,y,z)

Lengthy templates may result in awkward relation names under the latter method.

Templates do have their advantages. Omega-1.5 is very weak when adjectives are desired. For instance, in the Zoo example, the following relations were defined:

long_neck

tawny_color
dark_spots

The adjectives could not stand alone. A template approach could easily resolve this:

_ has long neck
_ has tawny color
_ has dark spots

Templates are also very useful for adding units to numbers in a tuple. This weakness in Omega-1.5 is discussed in the next section. A final shortcoming of the Omega-1.5 grammar is that Omega-1.5 indirectly requires knowledge of the predicate logic syntax in order to program effectively. This knowledge would not be needed with the template approach.

Perhaps a combined approach might provide an ideal solution for the pseudo-natural notation. Features of the template approach and the Omega-1.5 approach could be synthesized into a common framework. This would be an excellent topic for further research.

2. Modifications and Extensions

The programming and implementation experience with Omega-1.5 has suggested several design modifications to the grammar. These conditions will be addressed as recommended areas for additional study.

a. Modifications

In Omega-1.5, a variable is bound if it is defined in the directory structure. If it is not defined, it is considered free. The distinction is made when the rules are activated. This requires the programmer to remember which names have been previously defined. It also requires the programmer to remember reserved words. This

could be avoided if there was a syntactical distinction between free and bound variables. In the McArthur implementation, it was suggested that free variables begin with lower case letters, and bound variables begin with upper case letters. This is not practical in Omega-1.5. Other conventions must be explored. An extra character could be added at the end of a free variable for example (list versus list@), but this would diminish the readability of the code.

The use of values is a second area which needs to be reexamined. Specific emphasis should be on the use of numbers. In many cases, the units of the numbers are desired for added readability. For instance, the relation:

dimensions (rectangle,3,2).

would be written in Omega-1.5 as:

The rectangle has dimensions of 3 by 2.

It would be desirable to be able to write:

The rectangle has dimensions of 3 feet by 2 feet.

Omega-1.5 does not permit this, however. The best it can do is:

```
The rectangle has dimensions of 3 !feet! by
                                    2 !feet!.
```

In this example, the units are described by in-line comments.

A third observation involves the mailbox construct. By convention, the mailbox relation for Omega-1 procedure calls is placed as the first argument in the relation's tuple. For example, the procedure **pushed** is written in the premise as:

pushed(a,x,s)

and in the conclusion as:

pushed{x,s}

In Omega-1.5, pushed would be:

premise: a user has pushed an item on the stack
conclusion: an item (procedure) is pushed on the stack

Although this is easily accomplished, it would be even simpler in many cases if the mailbox was placed as the rightmost argument in the relation tuple. Therefore pushed could be written as:

premise: an item is pushed on the stack by a user
conclusion: an item (procedure) is pushed on the stack

Here, the basic structure does not change. The only modification is the appending of a prepositional phrase to the premise side.

b. Extensions

There are two extensions to the Omega-1.5 grammar that should be examined in future studies. Currently, only one argument (the subject) is permitted in front of the auxiliary verb and the relation name. While translating a previously written interpreter/pretty printer, the following relation was encountered:

Eval (Template,node)

It would be nice to be able to translate this as:

The template for the node is evaluated

This would call for two arguments in front of the auxiliary verb and relation name, however. Permitting multiple

arguments in front of the relation name would not be a very difficult change to implement. It would only require a more sophisticated mechanism for inverting the numerous arguments with the relation name.

Just as multiple arguments should be permitted in front of the auxiliary verb and relation name, no arguments should also be permitted. This would allow procedure calls (and relations) with null tuples. A procedure call could therefore be:

```
Do terminate (prccedure).
```

This would be translated as:

```
terminate{}.
```

Again, this change would be easy to implement without causing major parsing problems.

B. REMARKS ON THE OMEGA CONCEPT

The Omega language offers an ideal framework for many problems. The key ccmponent in Omega is its production rule model. Omega favors problems that can be decomposed into cause/effect productien rules. The production rules should be independent with minimal communication required between the rules.

The independence between the rules is very important. Rules must also be written so that they don't delete information that may be required by another rule. This prcblem was experienced during the development of several rule-based systems. Consider the following example in Omega-1 (Mary, John, food, and wine are defined objects):

```
define {root, "rules",
<< if *likes(Mary,x), likes(John,x) ->
                                display{"red"};
   if *likes(John,Mary), likes(Mary,wine) ->
```

```
                                        display{"blue"}
    >>}.
    act{rules}.
```

If the user enters the following in the command mode:

```
likes(Mary,wine).
likes(John,wine).
likes(John,Mary).
```

only rule 1 will fire when in fact the user has entered proper information to fire both rules. The problem is that one inquiry in the premise must be deleted in order to prevent the rule from continuously firing. There are programming methods to preclude this condition from occurring, but they are not trivial. Perhaps a mechanism can be developed so that (if desired) an active rule would fire only once in the same state.

One other recommended extension is a query capability similar to the question in Prolog. Currently in the command mode, a user can find out if John is the brother of Mary by entering:

```
if brother(John,Mary) -> display{"yes"}.
```

It would be nice if the user could instead enter:

```
?brother(John,Mary)
or ?brother(John,x)
```

It should not be a difficult change to make. The challenge would be to develop a suitable representation for the Omega-1.5 grammar. A quick solution would be to terminate the statement with a question mark instead of a period:

```
John is the brother of Mary?
```

In this case, Omega would respond with either a yes or a no. The second example (?brother(John,x)) could be written:

John is the brother of whom?

where whom represents the uninstantiated variable x. Here, Omega would either respond with a binding for x or with Nil.

C. CONCLUSIONS

The Omega system is an object-oriented programming language predicated upon a simulation paradigm. Omega differs from conventional languages in that it lacks such common structures as variables, assignment statements, and most control structures. Key to the Omega design is the event-oriented transaction rule. This is used to describe state changes in the system.

This thesis has described a stylized natural language grammar for the Omega programming language. This grammar is offered as an alternative to the predicate logic notation of Omega-1. The major advantages of the naturalized style include easier reading and semantic understanding of the code.

Principal objectives in the design of the Omega-1.5 grammar were simplicity and flexibility. The simplicity of the design is evidenced by Omega-1.5's ability to be parsed by traditional syntactic parsing techniques. There is no need for the conceptual dependency parsing usually associated with more sophisticated grammars that attempt to closely parallel true natural language. Additionally, Omega-1.5 avoids the semantic ambiguities that would be incurred if a true natural language grammar could be implemented.

A prototype translator was built to test the feasibility of the Omega-1.5 design. A major goal in the translator was

to examine extensible options for the grammar. Extensibility is essential for flexibility in coding applications.

Our final area of emphasis was the development of sample application programs. These programs were developed to demonstrate the Omega-1.5 grammar and to suggest potential application areas for the Omega language. Possible applications range from rule-based systems to programming environments. It is hoped that the programming style of these programs will assist future Omega programming efforts and also serve as a focal point for additional research with the Omega language.

APPENDIX A
SYNTAX OF OMEGA-1.5

("?" denotes optional)

```
session          =    '.'
                  |   statement_list '.'

statement_list   =    statement
                  |   statement '.' statement_list

statement        =    cpd_rule
                  |   fn_definition

cpd_rule         =    rule
                  |   rule 'else' cpd_rule

rule             =    start_clause cause 'then' effect
                  |   start_clause cause 'then'
                  |   'then' effect
                  |   effect

start_clause     =    'if'
                  |   'when'

cause            =    conditions

conditions       =    condition
                  |   constraint
                  |   condition ',' 'and'? conditions
                  |   constraint ',' 'and'? conditions

condition        =    'given'? inquiry

inquiry          =    subject relation_phrase arguments

constraint       =    expression

effect           =    transactions
```

```
transactions     =   transaction
                 |   transaction ',' 'and'?
                                    transactions

transaction      =   predication
                 |   expression
                 |   seq_block

predication      =   subject relation_phrase
                                    arguments

fn_definition    =   'function' fn_head ':' cpd_expr

fn_head          =   word_phrase? '[' arguments1 ']'

arglist          =   expression
                 |   expression noise_preps arglist

arguments        =   /* empty */
                 |   arglist
                 |   expression ':' expression

arguments1       =   /* empty */
                 |   list
                 |   expression ':' expression

list             =   expression
                 |   expression ','? list

seq_block        =   'begin' s_list 'end_block'

s_list           =   statement
                 |   statement ';' s_list

subject          =   expression
                 |   expression ':' expression

expression       =   word_phrase
                 |   relation_phrase
                 |   constant
```

```
                |   call
                |   fn_application
                |   rule_denotation
                |   unop
                |   binop
                |   noise_preps? noise_word? '('
                        cpd_expr ')'
                |   noise_preps? noise_word?
                        list_starter list
                |   noise_preps? noise_word?
                        list_starter expression
                        ':' expression
                |   noise_preps? noise_word?
                        list_starter

list_starter    =   'the_list'

fn_application  =   word_phrase '(' 'function'
                        ')' arguments
                |   subject '(' 'predicate' ')'
                        relation_phrase arguments

call            =   subject '(' 'procedure' ')'
                        relation_phrase arguments

noise_verb      =   'is' | 'are' | 'has' | 'do'

noise_word      =   'a' | 'an' | 'the'

noise_prep      =   'of' | 'to' | 'for' | 'as'
                |   'at' | 'with' | 'along' | 'on'
                |   'from' | 'plus' | 'and'

noise_preps     =   noise_prep
                |   noise_prep noise_preps

word_phrase     =   noise_preps? noise_word? word

rule_denotation =   'rules' s_list ';'? 'end_rules'
```

```
unop              =    '+' expression
                  |    '-' expression
                  |    '¬' expression

binop             =    expression '+' expression
                  |    expression '-' expression
                  |    expression '*' expression
                  |    expression '/' expression
                  |    expression '%' expression
                  |    expression '=' expression
                  |    expression '<' expression
                  |    expression '>' expression
                  |    expression '¬=' expression
                  |    expression '<=' expression
                  |    expression '>=' expression
                  |    expression '&' expression
                  |    expression '|' expression

relation_phrase   =    noise_verb 'not'? noise_verb?
                                        word_phrase

word              =    IDENTIFIER

constant          =    STR_CON
                  |    INT_CON
                  |    NIL

cpd_expr          =    cond_expr
                  |    cond_expr 'else' cpd_expr

cond_expr         =    expression
                  |    start_clause constraint 'then'
                                        expression
```

APPENDIX B

COMPARISON OF OMEGA-1.5 AND OMEGA-1 CONSTRUCTS

I. General:

Omega-1.5	Omega-1
then	->
given	*
function	fn
rules	<<
end_rules	>>
begin	{
end_block	}
function	[
predicate	[
procedure	{
not	¬

II. Built-in Procedures:

Omega-1.5	Omega-1
activate	act
displayed	display
displayed_with_return	displayn
relation	newrel{}
object	newobj{}

III. Built-in Functions:

Omega-1.5	Omega-1
an_integer	IsInt

a_string	IsStr
a_list	IsList
an_object	IsObj
a_relaticn	IsRel
converted_to_string	int_str
appending	cons
bound_to_object	objval

APPENDIX C

COMPARATIVE APPLICATIONS: OMEGA-1.5, OMEGA-1

```
!*****************************************
*                                       *
*          PDA -- Omega-1.5             *
*                                       *
*****************************************!

!
  PDA is a program tc implement
  a deterministic pushdown autcmaton (J)
  where J = ({state1,state2},{0,1,c},{red,blue,green},
                        o,state1,red,empty stack).

  J accepts {wcw^R| w in (0 + 1)*} by empty stack.

  This program was developed to show the implementation
  of a stack abstract data type
!

!  Rules to implement stack abstract data type  !

"Pushed" (procedure) is defined as a relation.
"Popped" (procedure) is defined as a relation.
"Contents" (procedure) is defined as a relation.
"Available" (procedure) is defined as a relation.
"Requested" (procedure) is defined as a relation.
"Destroy" (procedure) is defined as a relation.
"Top_item" (procedure) is defined as a relation.
"Cleared" (procedure) is defined as a relation.

"Does" (procedure) is defined as a noise_verb.
"Sent" (procedure) is defined as a noise_verb.
"Being" (procedure) is defined as a noise_verb.
"Want" (procedure) is defined as a noise_verb.
```

```
"New_stack" (procedure) is defined as an object.

!  Put an object in the available relation  !

An object is available.

!  The stack rules  !

Stack_rules (procedure) are defined as
  Rules
    If given a user has pushed an item on a stack, and
      given a list is the contents of the stack
    then
      the stack is sent to the user, and
      the appending (function) of the item with the list
        is the contents of the stack;

    If given a user has popped a stack, and
      Nil is the contents of the stack
    then
      Nil is sent to the user, and
      "Stack underflow." (procedure) is displayed

    Else if given a user has popped a stack, and
      given a list is the contents of the stack
    then
      the first (function) of the list is sent to
        the user, and
      the rest (function) of the list is the contents
        of the stack;

    If given a user does want the top_item of the
      stack, and
      a list is the contents of the stack
    then the first (function) of the list is sent to
      the user;

    If given a user has requested a new_stack, and
```

```
      given a stack is available
    then
      the stack is sent to the user, and
      Nil is the contents of the stack

    Else if given a user has requested a new_stack
    then
      "No stack available." (procedure) is displayed;

    If given a user does destroy a stack, and
      given a list is the contents of the stack
    then
      the list is sent to the user;

    If given a user has cleared a stack
    then
      the stack is sent to the user, and
      Nil is the contents of the stack;
  end_rules.

The stack_rules (procedure) are activated.

!  Request a stack called pda_stack  !
"PDA_stack" (procedure) is defined as a new_stack
   (procedure) being requested.

!  Rules for the PDA  !
"Input" (procedure) is defined as a relation.
"State1" (procedure) is defined as a relation.
"State2" (procedure) is defined as a relation.

"Current_state" (procedure) is defined as an object.
"Red" (procedure) is defined as an object.
"Green" (procedure) is defined as an object.
"Blue" (procedure) is defined as an object.
"C" (procedure) is defined as an object.

!  Initial state  !
Red (procedure) is pushed on the pda_stack.
```

```
The current_state is state1.

"PDA_rules" (procedure) are defined as
  Rules
    If given a list is input,
      the list ¬= Nil,
      the current_state is state1, and
      the first (function) of the list = 0
    then
      blue (procedure) is pushed on the pda_stack, and
      the rest (function) of the list is the input

    Else if given a list is input,
      the list ¬= Nil,
      the current_state is state1, and
      the first (function) of the list = 1
    then
      green (procedure) is pushed on the pda_stack, and
      the rest (function) of the list is the input

    Else if given a list is input,
      the list ¬= Nil,
      the current_state is state1, and
      the first (function) of the list = c
    then
      the current_state is not state1,
      the current_state is state2, and
      the rest (function) of the list is the input

    Else if given a list is input,
      the list ¬= Nil,
      the current_state is state2,
      the first (function) of the list = 0, and
      the pda_stack (procedure) has a top_item = blue
    then
      the pda_stack (procedure) is popped, and
      the rest (function) of the list is the input
```

```
    Else if given a list is input,
      the list ¬= Nil,
      the current_state is state2,
      the first (function) of the list = 1, and
      the pda_stack (procedure) has a top_item = green
    then
      the pda_stack (procedure) is popped, and
      the rest (function) of the list is the input

    Else if given a list is input,
      the list = Nil,
      given the current_state is state2, and
      the pda_stack (procedure) has a top_item = red
    then
      the pda_stack (procedure) is popped,
      "accept" (procedure) is displayed,
!   return to initial state  !
      red (procedure) is pushed on the pda_stack, and
      the current_state is state1

    Else if given a list is input, and
      given the current_state is state2
    then
      "Nc" (procedure) is displayed,
!   return to initial state  !
      the pda_stack (procedure) is cleared,
      red (procedure) is pushed on the pda_stack, and
      the current_state is state1

    Else if given a list is input
    then
      "No" (procedure) is displayed,
!   return to initial state  !
      the pda_stack (procedure) is cleared, and
      red (procedure) is pushed on the pda_stack;
```

```
  end_rules.
```

The pda_rules (procedure) are activated.

```
!*****************************************
!                                        *
!            PDA -- Cmega-1              *
!                                        *
!*****************************************

!  Rules to implement stack abstract data type

define {root,"pushed",newrel{}}.
define {root,"popped",newrel{}}.
define {root,"contents",newrel{}}.
define {root,"available",newrel{}}.
define {root,"requested",newrel{}}.
define {root,"destroy",newrel{}}.
define {root,"top_item",newrel{}}.
define {root,"cleared",newrel{}}.
define {root,"new_stack",newobj{}}.

!  Put an object in the available relation.
available(newobj{}).

!  The stack rules
define{root,"Stack_rules",
<<
  if *pushed(user,item,stack), *contents(list,stack) ->
                  user(stack),
                  contents(cons[item,list],stack);

  if *popped(user,stack), contents(Nil,stack) ->
                  user(Nil),
                  display{"Stack underflow."}
```

```
  else if *popped(user,stack), *contents(list,stack) ->
                  user(first[list]),
                  contents(rest[list],stack);

  if *top_item(user,stack), contents(list,stack) ->
                  user(first[list]);

  if *requested(user,new_stack), *available(stack) ->
                  user(stack),
                  contents(Nil,stack)

  else if *requested(user,new_stack) ->
                  display{"No stack available."};

  if *destroy(user,stack), *contents(list,stack) ->
                  user(list);

  if *cleared(user,stack) ->
                  user(stack),
                  contents(Nil,stack);
>>}.

act{Stack_rules}.

!  Establish a stack called pda_stack.
define{root,"pda_stack",requested{new_stack}}.

!  Rules for the PDA
define {root,"input",newrel{}}.
define {root,"state1",newrel{}}.
define {root,"state2",newrel{}}.
define {root,"current_state",newrel{}}.
define {root,"red",newobj{}}.
define {root,"green",newobj{}}.
define {root,"blue",newobj{}}.
define {root,"c",newobj{}}.

!  Initial state
pushed{red,pda_stack}.
```

```
state1(current_state).

define{root,"pda_rules",
<<
  if *input(x), x¬=Nil, state1(current_state),
                first[x]=0 ->
                pushed{blue,pda_stack},
                input(rest[x])

  else if *input(x), x¬=Nil, state1(current_state),
                first[x]=1 ->
                pushed{green,pda_stack},
                input(rest[x])

  else if *input(x), x¬=Nil, state1(current_state),
                first[x]=c ->
                ¬state1(current_state),
                state2(current_state),
                input(rest[x])

  else if *input(x), x¬=Nil, state2(current_state),
                first[x]=0,
                top_item{pda_stack}=blue ->
                popped{pda_stack},
                input(rest[x])

  else if *input(x), x¬=Nil, state2(current_state),
                first[x]=1,
                top_item{pda_stack}=green ->
                popped{pda_stack},
                input(rest[x])

  else if *input(x), x=Nil, *state2(current_state),
                top_item{pda_stack}=red ->
                popped{pda_stack},
                display{"accept"},
                !  return to initial state
                pushed{red,pda_stack},
```

```
                    state1(current_state)

  else if *input(x), *state2(current_state) ->
                    display{"no"},
                    !  return to initial state
                    cleared{pda_stack},
                    pushed{red,pda_stack},
                    state1(current_state)

  else if *input(x) ->
                    display{"no"},
                    !  return to initial state
                    cleared{pda_stack},
                    pushed{red,pda_stack}
>>}.

act{pda_rules}.



!*****************************************
!                                        *
!           Logic5 -- Omega-1            *
!                                        *
!*****************************************

!  Logic in Omega
!    Propositions represented by values (lists)
!    Concepts (objects of knowledge, namely
!            individuals, 'is's and 'imp's)
!            can be represented by either values
!            or objects.
!    They are represented by values (strings) in
!            this example.
!    Proposition identification by pattern matching.

!  Cognitive functions
```

```
define {root,"Asks",newrel{}};
define {root,"Knows",newrel{}};
define {root,"Inquire",newrel{}};

!  Kinds of propositions
define {root,"isa","isa"};
define {root,"imp","imp"};
define {root,"not","not"};

define {root,"Logic5Rules",
<<
!  Inquiry management
if *Inquire(s,p), Knows(s,p) -> displayn{"yes"};
if *Inquire(s,p), Knows(s, [not,p]) -> displayn{"no"};

if Inquire(s,p), ¬Knows(s,p), ¬Knows(s, [not,p]),
   ¬Asks(s,p), ¬Asks(s, [not,p])
-> Asks(s,p), Asks(s, [not,p]);

if Knows(s,p), *Inquire(s,p) -> displayn{"yes"};
if Knows(s,[not,q]), *Inquire(s,q) -> displayn{"no"};

!  Deductive rules
if *Asks(s,[isa,x,p]), Knows(s,[imp,q,p]),
         Knows(s,[isa,x,q])
-> Knows(s,[isa,x,p]);

if Asks(s,[isa,x,p]), Knows(s,[imp,q,p]),
         ¬Knows(s,[isa,x,q]), ¬Asks(s,[isa,x,q])
-> Asks(s,[isa,x,q])
>>}.

!  Concepts
define {root,"man","man"};
define {root,"dog","dog"};
define {root,"animal","animal"};
define {root,"mortal","mortal"};
define {root,"Soc","Soc"};
```

```
define {root,"Fido","Fido"}.

!  Knowledge
Knows(self,[isa,Soc,man]);
Knows(self,[imp,man,animal]);
Knows(self,[imp,animal,mortal]);
Knows(self,[imp,dog,animal]);
KNows(self,[isa,Fido,dog]).

act{Logic5Rules}.



!*****************************************
*                                       *
*         Logic5 -- Omega-1.5           *
*                                       *
*****************************************!


!
    Propositions are represented by values (lists).
    Concepts are represented by objects in this example.
    Proposition identification by pattern matching.
!

!   Cognitive functions   !
"Ask" (procedure) is defined as a relation.
"Know" (procedure) is defined as a relation.
"Inquiring" (procedure) is defined as a relation.

"Asked" (procedure) is defined for ask.

!   Kinds of propositions   !
"Is_a" (procedure) is defined as an object.
"Implies" (procedure) is defined as an object.
"Negation" (procedure) is defined as an object.
```

```
"I" (procedure) is defined as an object.
"Am" (procedure) is defined as a noise_verb.
"Will" (procedure) is defined as a noise_verb.
"Have" (procedure) is defined as a noise_verb.
"About" (procedure) is defined as a noise_prep.
"Proposition_that" (procedure) is defined as
   a list_starter.
"Assertion" (procedure) is defined as a list_starter.

"Logic_rules" (procedure) are defined as
  Rules
!   Inquiry management   !
    If given I am inquiring about a proposition, and
      I do know the proposition
    then
      "yes" (procedure) is displayed;

    If given I am inquiring about a proposition, and
      I do know the assertion of the negation of the
      proposition
    then
      "nc" (procedure) is displayed;

    If I am inquiring about a proposition,
      I do not know the proposition,
      I do not know the assertion of the negation
        of the proposition,
      I have not asked about the proposition, and
      I have not asked about the assertion of the
        negation of the proposition
    then
      I will ask about the proposition, and
      I will ask about the assertion of the
        negation of the proposition;

    If I do know about a proposition, and
```

```
      given I am inquiring about the proposition
    then
      "yes" (procedure) is displayed;

    If I do know about the assertion of the
      negation of a proposition, and
      given I am inquiring about the proposition
    then
      "nc" (procedure) is displayed;

!   Deductive rules   !
    If given I have asked about the proposition_that
          object1 is_a object2,
      I do know the proposition_that
          object3 implies object2, and
      I do know the proposition_that object1 is_a
          object3
    then
      I do know the proposition_that object1 is_a
          object2;

    If I have asked about the proposition_that
          object1 is_a object2,
      I do know the proposition_that object3 implies
          object2,
      I do not know the proposition_that object1 is_a
          object3, and
      I have not asked about the proposition_that
          object1 is_a object3
    then
      I will ask about the proposition_that object1
          is_a object3;
  end_rules.

!   Concepts   !
"Man" (procedure) is defined as an object.
"Dog" (procedure) is defined as an object.
```

```
"Animal" (procedure) is defined as an object.
"Mortal" (procedure) is defined as an object.
"Soc" (procedure) is defined as an object.
"Fido" (procedure) is defined as an object.

!   Knowledge   !
I do know the proposition_that Soc is_a man.
I do know the proposition_that man implies animal.
I do know the proposition_that animal implies mortal.
I do know the proposition_that dog implies animal.
I do know the proposition_that Fido is_a dog.

The logic_rules (procedure) are activated.



!*****************************************
!                                        *
!      Logic5 -- Omega-1 translated      *
!                                        *
!*****************************************

!   Cognitive functions
defined {"ask",newrel{}}.
defined {"know",newrel{}}.
defined {"inquiring",newrel{}}.

defined {"asked",ask}.

!   Kinds of propositions
defined {"is_a",newobj{}}.
defined {"implies",newobj{}}.
defined {"negation",newobj{}}.

defined {"I",newobj{}}.

defined {"logic_rules",
<<
```

```
!   Inquiry management
if *inquiring(I,propcsition), know(I,proposition) ->
      display{"yes"};
if *inquiring(I,proposition),
      know(I,[negaticn,proposition]) -> display{"no"};

if inquiring(I,proposition), ¬know(I,proposition),
      ¬know(I,[negaticn,proposition]),
      ¬asked(I,proposition),
      ¬asked(I,[negaticn,proposition])
-> ask(I,proposition), ask(I,[negation,proposition]);

if know(I,proposition), *inquiring(I,proposition)
      -> display{"yes"};
if know(I,[negation,proposition]),
      *inquiring(I,prcposition)
      -> display{"no"};

!   Deductive rules
if *asked(I,[object1,is_a,object2]),
  know(I,[object3,implies,object2]),
  know(I,[object1,is_a,object3])
-> kncw(I,[object1,is_a,object_2]);

if asked(I,[object1,is_a,object2]),
  know(I,[object3,implies,object2]),
  ¬kncw(I,[object1,is_a,object3]),
  ¬asked(I,[object1,is_a,object3])
-> ask(I,[object1,is_a,object3])
>>}.

!   Ccncepts
defined {"man",newobj{}}.
defined {"dog",newobj{}}.
defined {"animal",newcbj{}}.
defined {"mortal",newcbj{}}.
defined {"Soc",newobj{}}.
```

```
defined {"Fido",newobj{}}.

!   Knowledge
know(I,[Soc,is_a,man]).
know(I,[man,implies,animal]).
know(I,[animal,implies,mortal]).
know(I,[dog,implies,animal]).
know(I,[Fido,is_a,dog]).

act{logic_rules}.



!*****************************************
!                                        *
!      Towers of Hanoi -- Omega-1        *
!                                        *
!*****************************************

!  Define the relations
define {root,"Hanoi",newrel{}}.
define {root,"HanoiAux",newrel{}}.

!  The rules
define {root,"HanoiRules",
<<
if *Hanoi(a,n) ->
        HanoiAux(a, n, "A", "C", "B");

if *HanoiAux(a, 1, from, to, aux) ->
        {
        display{"Move disk 1 from peg "};
        display{from};
        display{" to peg "};
        displayn{to};
        a("")
        }
```

```
else if *HanoiAux(a, n, from, to, aux) ->
        {
        HanoiAux{n-1, from, aux, to};
        display{"Move disk "};
        display{n};
        display{" from peg "};
        display{from};
        display{" to peg "};
        displayn{to};
        HanoiAux{n-1, aux, to, from};
        a("")
        }
>>}.

act{HanoiRules}.
```

```
!*****************************************
*                                       *
*      Towers of Hanoi -- Omega-1.5     *
*                                       *
*****************************************!

"Input" (procedure) is defined as a relation.
"Moved" (procedure) is defined as a relation.
"PegA" (procedure) is defined as an object.
"PegB" (procedure) is defined as an object.
"PegC" (procedure) is defined as an object.

"Must" (procedure) is defined as a noise_verb.
"Be" (procedure) is defined as a noise_verb.
"Sent" (procedure) is defined as a noise_verb.
"Does" (procedure) is defined as a noise_verb.
```

```
"Hanoi_rules" (procedure) are defined as
  Rules
    If given a user dces input n_disks
    then n_disks must be moved from pegA to pegC
      with pegB;

    If given a user has moved 1 from the source_peg
      to the destination_peg with the auxiliary_peg
    then
      begin
        "Move disk 1 from peg " (procedure)
            is displayed;
         source_peg (procedure) is displayed;
         " to peg " (procedure) is displayed;
         destination_peg (procedure) is
            displayed_with_return;
         Nil is sent to the user
      end_block

    Else if given a user has moved the nth_disk from
      the source_peg to the destination_peg with the
      auxiliary_peg
    then
      begin
        The nth_disk-1 (procedure) must be moved from
          the source_peg to the auxiliary_peg with
          the destination_peg;
        "Move disk " (procedure) is displayed;
        nth_disk (procedure) is displayed;
        " from peg " (procedure) is displayed;
        source_peg (procedure) is displayed;
        " to peg " (procedure) is displayed;
        destination_peg (procedure) is
              displayed_with_return;
        The nth_disk-1 (procedure) must be moved from
          the auxiliary_peg to the destination_peg
```

```
          with the source_peg;
        Nil is sent to the user
      end_block;
  end_rules.
```

Hanoi_rules (procedure) are activated.

```
!*****************************************
*                                       *
*           Zoo -- Omega-1.5            *
*                                       *
*****************************************!
```

"Hair" (procedure) is defined as a relation.
"Mammal" (procedure) is defined as a relation.
"Give" (procedure) is defined as a relation.
"Feathers" (procedure) is defined as a relation.
"Bird" (procedure) is defined as a relation.
"Fly" (procedure) is defined as a relation.
"Lay" (procedure) is defined as a relation.
"Eat" (procedure) is defined as a relation.
"Carnivore" (procedure) is defined as a relation.
"Pointed_teeth" (procedure) is defined as a relation.
"Claws" (procedure) is defined as a relation.
"Forward_pointing" (procedure) is defined as a relation.
"Hoofs" (procedure) is defined as a relation.
"Ungulate" (procedure) is defined as a relation.
"Chew" (procedure) is defined as a relation.
"Even_toed" (procedure) is defined as a relation.
"Tawny_cclor" (procedure) is defined as a relation.
"Black_stripes" (procedure) is defined as a relation.
"Dark_spots" (procedure) is defined as a relation.
"Long_legs" (procedure) is defined as a relation.

```
"Long_neck" (procedure) is defined as a relation.
"White_cclor" (procedure) is defined as a relation.
"Black_and_white" (prccedure) is defined as a relation.
"Swim" (procedure) is defined as a relation.
"Good_flyer" (procedure) is defined as a relation.

"Eggs" (procedure) is defined as an object.
"Meat" (procedure) is defined as an object.
"Eyes" (procedure) is defined as an object.
"Cud" (procedure) is defined as an object.
"Animal" (procedure) is defined as an object.
"Milk" (procedure) is defined as an object.

"Does" (procedure) is defined as a noise_verb.

"Zoo_rules" (procedure) are defined as
  Rules
    !1!  if given the animal has hair then the animal
            is a mammal;

    !2!  if given the animal does give milk then the
            animal is a mammal;

    !3!  if given the animal has feathers then the
            animal is a bird;

    !4!  if given the animal does fly, and
            the animal does lay eggs
         then the animal is a bird;

    !5!  if given the animal is a mammal, and
            the animal does eat meat
         then the animal is a carnivore;

    !6!  if given the animal is a mammal,
            the animal has pointed_teeth,
            the animal has claws, and
            the animal has forward_pointing eyes
         then the animal is a carnivore;
```

```
!7!  if given the animal is a mammal, and
        the animal has hoofs
     then the animal is an ungulate;

!8!  if given the animal is a mammal, and
        the animal does chew cud
     then the animal is an ungulate, and
        the animal is even_toed;

!9!  if given the animal is a carnivore,
        the animal has a tawny_color, and
        the animal has dark_spots
     then "The animal is a cheetah" (procedure)
        is displayed;

!10! if given the animal is a carnivore,
        the animal has a tawny_color, and
        the animal has black_stripes
     then "The animal is a tiger" (procedure)
        is displayed;

!11! if given the animal is an ungulate,
        the animal has long_legs,
        the animal has a long_neck,
        the animal has a tawny_color, and
        the animal has dark_spots
     then "The animal is a giraffe" (procedure)
        is displayed;

!12! if given the animal is an ungulate,
        the animal has a white_color, and
        the animal has black_stripes
     then "The animal is a zebra" (procedure)
        is displayed;

!13! if given the animal is a bird,
        the animal does not fly,
```

```
            the animal has long_legs,
            the animal has a long_neck, and
            the animal is black_and_white
         then "The animal is an ostrich" (procedure)
            is displayed;

    !14! if given the animal is a bird,
            the animal does not fly,
            the animal does swim, and
            the animal is black_and_white
         then "The animal is a penguin" (procedure)
            is displayed;

    !15! if given the animal is a bird, and
            the animal is a good_flyer
         then "The animal is an albatross" (procedure)
            is displayed;

  end_rules.

The zoo_rules (procedure) are activated.



!*****************************************
!                                        *
!       Zoo -- Omega-1 translation       *
!                                        *
!*****************************************

defined {"hair",newrel{}}.
defined {"mammal",newrel{}}.
defined {"give",newrel{}}.
defined {"feathers",newrel{}}.
defined {"bird",newrel{}}.
defined {"fly",newrel{}}.
defined {"lay",newrel{}}.
```

```
defined {"eat",newrel{}}.
defined {"carnivore",newrel{}}.
defined {"pointed_teeth",newrel{}}.
defined {"claws",newrel{}}.
defined {"forward_pointing",newrel{}}.
defined {"hoofs",newrel{}}.
defined {"ungulate",newrel{}}.
defined {"chew",newrel{}}.
defined {"even_toed",newrel{}}.
defined {"tawny_color",newrel{}}.
defined {"black_stripes",newrel{}}.
defined {"dark_spots",newrel{}}.
defined {"long_legs",newrel{}}.
defined {"long_neck",newrel{}}.
defined {"white_color",newrel{}}.
defined {"black_and_white",newrel{}}.
defined {"swim",newrel{}}.
defined {"good_flyer",newrel{}}.

defined {"eggs",newobj{}}.
defined {"meat",newobj{}}.
defined {"eyes",newobj{}}.
defined {"cud",newobj{}}.
defined {"animal",newobj{}}.
defined {"milk",newobj{}}.

defined {"zoo_rules",
<<
  !1
   if *hair(animal) -> mammal(animal);

  !2
   if *give(animal,milk) -> mammal(animal);

  !3
   if *feathers(animal) -> bird(animal);
```

```
!4
 if *fly(animal), lay(animal,eggs) -> bird(animal);

!5
 if *mammal(animal), eat(animal,meat) ->
    carnivore(animal);

!6
 if *mammal(animal), pointed_teeth(animal),
    claws(animal), forward_pointing(animal,eyes) ->
    carnivore(animal);

!7
 if *mammal(animal), hoofs(animal) ->
    ungulate(animal);

!8
 if *mammal(animal), chew(animal,cud) ->
    ungulate(animal), even_toed(animal);

!9
 if *carnivore(animal), tawny_color(animal),
    dark_spots(animal) ->
    display{"The arimal is a cheetah"};

!10
 if *carnivore(animal), tawny_color(animal),
    black_stripes(animal) ->
    display{"The animal is a tiger"};

!11
 if *ungulate(animal), long_legs(animal),
    long_neck(animal), tawny_color(animal),
    dark_spots(animal) ->
    display{"The animal is a giraffe"};

!12
 if *ungulate(animal), white_color(animal),
    black_stripes(animal) ->
```

```
      display{"The animal is a zebra"};

  !13
   if *bird(animal), ¬fly(animal),
      long_legs(animal), long_neck(animal),
      black_and_white(animal) ->
      display{"The animal is an ostrich"};

  !14
   if *bird(animal), ¬fly(animal),
      swim(animal), black_and_white(animal) ->
      display{"The animal is a penguin"};

  !15
   if *bird(animal), good_flyer(animal) ->
      display{"The animal is an albatross"};
>>}.

act{zoo_rules}.



!*****************************************
!                                        *
!           PI-1 -- Crega-1              *
!                                        *
!*****************************************

!   Rules and associated definitions for
!   an arithmetic expression language.

!   Relations

! Prcgram Structure Relations
define {root,"Appl",newrel{}};
define {root,"Op",newrel{}};
define {root,"Left",newrel{}};
define {root,"Right",newrel{}};
```

```
define {root,"Con",newrel{}};
define {root,"Litval",newrel{}};

! Evaluation Relations

define {root,"Eval",newrel{}};
define {root,"Check",newrel{}};
define {root,"Value",newrel{}};
define {root,"Meaning",newrel{}};
define {root,"Explanation",newrel{}};

! Unparser Relations

define {root,"Unparse",newrel{}};
define {root,"Image",newrel{}};
define {root,"Template",newrel{}};

! Command Interpreter Relations

define {root,"Command",newrel{}};
define {root,"Argument",newrel{}};
define {root,"Root",newrel{}};
define {root,"Undef",newrel{}};
define {root,"CurrentNode",newrel{}};

define {root,"EvalPending",newrel{}};
define {root,"ShowPending",newrel{}};
define {root,"CreateAppl",newrel{}};
define {root,"CreateRoot",newrel{}};
define {root,"Script",newrel{}};
define {root,"PendScript",newrel{}};

! Functions

fn Id [x]: x;
fn Sum [x,y]: x + y;
fn Dif [x,y]: x - y;
fn Product [x,y]: x * y;
fn Quotient [x,y]:
```

```
    if y = 0 -> ["error", 1]
    else x / y;

fn IsErrorcode [w]:
    if ¬IsList[w] | w = Nil -> Nil
    else first[w] = "error";

fn upSum [x,y]: "(" + x + "+" + y + ")";
fn upDif [x,y]: "(" + x + "-" + y + ")";
fn upProd [x,y]: "(" + x + "x" + y + ")";
fn upQuot [x,y]: "(" + x + "/" + y + ")".

! Built-in Tables

Meaning (Sum,"+");
Meaning (Dif,"-");
Meaning (Product,"x");
Meaning (Quotient,"/");
Meaning (Id,"lit");

Template (upSum,"+");
Template (upDif,"-");
Template (upProd,"x");
Template (upQuot,"/");
Template (int_str,"lit");

Explanation ("incomplete program", ["error",0]);
Explanation ("divisicn by zero", ["error",1]).

!   the Rules

define {root,"PI1Rules",
<<
! Evaluator Rules

! Constant nodes

if *Eval(e), Con(e), Iitval(v,e), Meaning(f,"lit")
-> Value(f[v],e);
```

```
! Appl nodes

if *Eval(e), Appl(e), Left(x,e), Right(y,e)
-> Eval(x), Eval(y);

if *Value(u,x), *Value(v,y),
   Appl(e), Op(n,e), Left(x,e), Right(y,e),
   Meaning(f,n)
-> Check(f[u,v],e);

! Error Checking

if *Check(w,e), ¬IsErrorcode[w]
-> Value(w,e);

if *Check(w,e), IsErrorcode[w],
   Explanation(s,w), *CurrentNode(q)
-> displayn{s}, CurrentNode(e);

! Unparser

! Constant Nodes

if *Unparse(e), Con(e), Litval(v,e),
   Template(f,"lit")
-> Image(f[v],e);

! Identifier nodes

! Appl nodes

if *Unparse(e), Appl(e), Left(x,e), Right(y,e)
-> Unparse(x), Unparse(y);

if *Image(u,x), *Image(v,y),
   Appl(e), Op(n,e), Left(x,e), Right(y,e),
   Template(f,n)
-> Image(f[u,v],e);

! Command Interpreter Rules

! evaluate Command
```

```
if *Command("evaluate"), CurrentNode(E)
-> Eval(E), EvalPending(E);

if *Value(V,E), *EvalPending(E)
-> displayn {V};

! return Command

if *Command("val"), *Argument(V), CurrentNode(E)
-> Value(V,E);

! show Command

if *Command("show"), CurrentNode(E)
-> Unparse(E), ShowPending(E);

if *Image(S,E), *ShowPending(E)
-> displayn {S};

! abort Command

if Command("abort"), *Eval(E) -> ;

if Command("abort"), *Value(V,E) -> ;

if Command("abort"), *Check(V,E) -> ;

if *Command("abort"), ¬Eval(E), ¬Value(V,E)
-> displayn{"aborted"};

! Handle incomplete program

if *Eval(E), Undef(E), *CurrentNode(Q)
-> displayn("Incomplete"), CurrentNode(E);

if *Unparse(E), Undef(E)
-> Image ("<expr>",E);

! Syntax Directed Editing

! in Command
```

```
if *Command("in"), *CurrentNode(E), Left(X,E)
-> CurrentNode(X), Command("show");

if *Command("out"), *CurrentNode(X), Left(X,E)
-> CurrentNode(E), Command("show");

if *Command("out"), *CurrentNode(Y), Right(Y,E)
-> CurrentNode(E), Command("show");

! next Command

if *Command("next"), *CurrentNode(X), Left(X,E),
   Right(Y,E)
-> CurrentNode(Y), Command("show");

! prev Command

if *Command("prev"), *CurrentNode(Y), Right(Y,E),
   Left(X,E)
-> CurrentNode(X), Command("show");

! delete command

if *Command("delete"), CurrentNode(E), *Con(E),
   *Litval(V,E)
-> Undef(E), Command("show");

if *Command("delete"), CurrentNode(E),
   *Appl(E), *Op(N,E), *Left(X,E),
   Right(Y,E)
-> Undef(E), Command("show");

if *Command("delete"), CurrentNode(E), Undef(E)
-> displayn("already deleted");

! # Command

if *Command("#"), *Argument(V), IsInt[V],
   CurrentNode(E), *Undef(E)
-> Con(E), Litval(V,E), Command("show");
```

```
if *Command("#"), *Argument(V), CurrentNode(E)
   ¬Undef(E)
-> displayn("defined node");

! +, -, x, / Commands

if *Command(op), member [op, ["+", "-", "x", "/"]],
   *CurrentNode(E), *Undef(E)
-> CreateAppl(op, E, newobj{}, newobj{});

if *CreateAppl(op,E,X,Y)
-> Appl(E), Op(op,E), Left(X,E), Right(Y,E),
   Undef(X), Undef(Y), CurrentNode(X);

if *Command(op), member [op, ["+", "-", "x", "/"]],
   CurrentNode(E), ¬Undef(E)
-> displayn("defined node");

! begin Command

if *Command("begin"), *CurrentNode(Q)
-> CreateRoot(newobj{});

if *CreateRoot(E)
-> Root(E), Undef(E), CurrentNode(E);

!  root Command

if *Command("root"), *CurrentNode(Q), Root(E)
-> CurrentNode(E), Command("show");

! Test Driver

if *Script(Nil) -> displayn{"Script completed"}

else if *Script(L), (first[L]="#" |
   first[L]="val")
-> { display {first [rest [L]]};
     displayn {first [L]};
     Command(first[L], Argument(first[rest[L]]),
     PendScript(rest[rest[L]]) }
```

```
else if *Script(L)
-> { displayn {first [L]};
     Ccmmand(first[L]), PendScript(rest[L]) };

if *PendScript(L), ¬Ccmmand(Q) -> Script(L)

>>}.

! activate the rules

act {PI1Rules}.
CurrentNode(Nil).
displayn{"PI-1 System loaded"}.



!*****************************************
*                                       *
*            PI-1 -- Cmega-1.5          *
*                                       *
*****************************************!

!
                PI-1
    Rules and associated definitions for
    an arithmetic expression language.
!

!  Relations  !

!  Prcgram structure relations  !

"Application" (procedure) is defined as a relation.
"Operator" (procedure) is defined as a relation.
"Left_argument" (procedure) is defined as a relation.
"Right_argument" (procedure) is defined as a relation.
"Constant" (procedure) is defined as a relation.
```

```
"Literal_value" (procedure) is defined as a relation.

! Evaluation relations !

"Evaluated" (procedure) is defined as a relation.
"Checked" (procedure) is defined as a relation.
"Value" (procedure) is defined as a relation.
"Meaning" (procedure) is defined as a relation.
"Explanation" (procedure) is defined as a relation.

! Unparser relations !

"Unparsed" (procedure) is defined as a relation.
"Image" (procedure) is defined as a relation.
"Template" (procedure) is defined as a relation.

! Command interpreter relations !

"Command" (procedure) is defined as a relation.
"Argument" (procedure) is defined as a relation.
"Root_node" (procedure) is defined as a relation.
"Undefined" (procedure) is defined as a relation.
"Current_node" (procedure) is defined as a relation.

"Pending_evaluation" (procedure) is defined as a relation.
"Shown" (procedure) is defined as a relation.
"New_application" (procedure) is defined as a relation.
"New_root" (procedure) is defined as a relation.
"Script" (procedure) is defined as a relation.
"Pending_script" (procedure) is defined as a relation.

! Functions !
function identity [x]: x.
function sum [x,y]: x + y.
function difference [x,y]: x - y.
function product [x,y]: x * y.
function quotient [x,y]:
     if y = 0 then the_list of the "error_code" and 1
     else x / y.
```

```
function error_code [W]:
   if W (predicate) is not a_list | W = Nil then Nil
   else the first (function) of W = "error_code".

function sum_template [x,y]: "(" + x "+" y + ")".
function difference_template [x,y]:
                "(" + x + "-" + y + ")".
function product_template [x,y]:
                "(" + x + "x" + y + ")".
function quotient_template [x,y]:
                "(" + x + "/" + y + ")".

!  Built-in tables  !

Sum is the meaning of "+".
Difference is the meaning of "-".
Product is the meaning of "x".
Quotient is the meaning of "/".
Identity is the meaning of "lit".

Sum_template is a template for "+".
Difference_template is a template for "-".
Product_template is a template for "x".
Quotient_template is a template for "/".
String_notation is a template for "lit".

"Incomplete program" is an explanation for the_list
      of error_code and 0.
"Division by zero" is an explanation for the_list of
      error_code and 1.

!  Noise words  !

"Must" (procedure) is defined as a noise_verb.
"Be" (procedure) is defined as a noise_verb.
"Being" (procedure) is defined as a noise_verb.
"Established" (procedure) is defined as a noise_verb.
```

```
"Will" (procedure) is defined as a noise_verb.
"Another" (procedure) is defined as a noise_verb.

!  The rules  !

"PI1_rules" (procedure) are defined as
Rules

!  Evaluator rules  !

!  Constant nodes  !

If given an expression is being evaluated,
   the expression is a constant,
   a number is the literal_value of the expression,
   and a lit_function is the meaning
   of "lit"
then the lit_function (function) of V is the value
   of the expression;

!  Application nodes  !

If given an expression is being evaluated,
   the expression is an application,
   node1 is the left_argument of the expression, and
   node2 is the right_argument of the expression
then node1 must be evaluated, and
   node2 must be evaluated;

If given value1 is the value of node1,
   given value2 is the value of node2,
   the expression is an application,
   a string is the operator of the expression,
   node1 is the left_argument of the expression,
   node2 is the right_argument of the expression, and
   an operator_function is the meaning of the string
then the operator_function (function) of value1 and
   value2 must be checked for the expression;
```

```
! Error checking !

If given an alleged_error is being checked for an
   expression, and
   the alleged_error (predicate) is not an error_code
then the alleged_error is the value of the expression;

If given an alleged_error is being checked for an
   expression,
   the alleged_error (predicate) is the error_code,
   a string is an explanation for the alleged_error,
   and given any_node is the current_node
then the string (procedure) is displayed_with_return,
   and the expression is the current_node;

! Unparser !

! Constant Nodes !

If given an expression is being unparsed,
   the expression is a constant,
   value1 is the literal_value of the expression, and
   a lit_function is a template for "lit"
then the lit_function (function) of value1 is the
   image of the expression;

! Identifier nodes !

! Application nodes !

If given an expression is being unparsed,
   the expression is an application,
   node1 is the left_argument of the expression, and
   node2 is the right_argument of the expression
then node1 must be unparsed, and
   node2 must be unparsed;

If given image1 is the image of node1,
   given image2 is the image of node2,
```

```
   the expression is an application,
   a string is the operator of the expression,
   node1 is the left_argument of the expression,
   node2 is the right_argument of the expression, and
   an operator_function is a template for the string
then the operator_function (function) of image1 and
   image2 is the image of the expression;

! Command interpreter rules !

! Evaluate command !

If given "evaluate" is the command, and
   an expression is the current_node
then the expression must be evaluated, and
   the expression is pending_evaluation;

If given value1 is the value of an expression, and
   the expression is pending_evaluation
then value1 (procedure) is displayed_with_return;

! Return command !

If given "val" is the command,
   given value1 is the argument, and
   an expression is the current_node
then value1 is the value of the expression;

! Show command !

If given "show" is the command, and
   an expression is the current_node
then the expression must be unparsed, and
   the expression will be shown;

If given a string is the image of an expression, and
   given the expression must be shown
then the string (procedure) is displayed_with_return;

! Abort command !
```

```
If "abort" is the command, and
   given an expression is being evaluated
then !do nothing! .

If "abort" is the command, and
   given a_value is the value of an expression
then !do nothing! .

If "abort" is the command, and
   given a_value is being checked for an expression
then !do nothing! .

If given "abort" is the command,
   an expression is not being evaluated, and
   a_value is not the value of the expression
then "aborted" (procedure) is displayed_with_return;

!  Handle incomplete program  !

If given an expression is being evaluated,
   the expression is undefined, and
   given any_node is the current_node
then "Incomplete" (procedure) is displayed_with_return,
   and the expression is the current_node;

If given an expression is being unparsed, and
   the expression is undefined
then "<expr>" is the image of the expression;

!  Syntax Directed Editing  !

!  in Command  !

If given "in" is the command,
   given an expression is the current_node, and
   node1 is the left_argument of the expression
then node1 is the current_node, and
   "show" is the command;
```

```
If given "out" is the command,
   given node1 is the current_node, and
   node1 is the left_argument of an expression
then the expression is the current_node, and
   "show" is the command;

If given "out" is the command,
   given node2 is the current_node, and
   node2 is the right_argument of an expression
then the expression is the current_node, and
   "show" is the command;

!  next Command  !

If given "next" is the command,
   given node1 is the current_node,
   node1 is the left_argument of an expression, and
   node2 is the right_argument of the expression
then node2 is the current_node, and
   "show" is the command;

!  prev Command  !

If given "prev" is the command,
   given node2 is the current_node,
   node2 is the right_argument of an expression, and
   node1 is the left_argument of the expression
then node1 is the current_node, and
   "show" is the command;

!  delete Command  !

If given "delete" is the command,
   an expression is the current_node,
   given the expression is a constant, and
   given a_value is the literal_value of the expression
then the expression is undefined, and
   "show" is the command;
```

```
If given "delete" is the command,
   an expression is the current_node,
   given the expression is an application,
   given a string is the operator of the expression,
   given node1 is the left_argument of the expression,
   and node2 is the right_argument of the expression
then the expression is undefined, and
   "show" is the command;

If given "delete" is the command,
   an expression is the current_node, and
   the expression is undefined
then "already deleted" (procedure) is
   displayed_with_return;

!  # Command  !

If given "#" is the command,
   given value1 is the argument,
   value1 (predicate) is an_integer,
   an expression is the current_node, and
   given the expression is undefined
then the expression is a constant,
   value1 is the literal_value of the expression, and
   "show" is the command;

If given "#" is the command,
   given value1 is the argument,
   an expression is the current_node, and
   the expression is not undefined
then "defined node" (procedure) is
   displayed_with_return;

!  +, -, x, / Commands  !

If given a string is the command,
   the string is a member of the_list
```

```
      of "+", "-", "x", and "/",
   given an expression is the current_node, and
   given the expression is undefined
then the expression is established as a
   new_application with a string and an object and
   another object;

If given an expression is a new_application with
   a string and node1 and node2
then the expression is an application,
   the string is the operator of the expression,
   node1 is the left_argument of the expression,
   node2 is the right_argument of the expression,
   node1 is undefined,
   node2 is undefined, and
   node1 is the current_node;

If given a string is the command,
   the string is a member of the_list
   of "+", "-", "x", and"/",
   an expression is the current_node, and
   the expression is not undefined
then "defined node" (procedure) is
   displayed_with_return;

!  begin Command  !

If given "begin" is the command, and
   given any_node is the current_node
then an object is established as a new_root;

If given an expression is a new_root
then the expression is a root_node,
   the expression is undefined, and
   the expression is the current_node;

!  root Command  !
```

```
If given "root" is the command,
   given any_node is the current_node, and
   an expression is the root_node
then the expression is the current_node, and
  "show" is the command;

!  Test driver  !

If given Nil is the script
then "Script completed" (procedure) is
   displayed_with_return

Else if given a list is the script, and
   (the first (function) of the list = "#" |
         the first (function) of the list = "val")

then
  begin
    the first (function) of the rest (function)
       of the list (procedure) is displayed;
    the first (function) of the list (procedure) is
       displayed_with_return;
    the first (function) of the list is the command,
    the first (function) of the rest (function) of the
       list is the argument, and
    the rest (function) of the rest (function) of the
       list is the pending_script
  end_block

Else if given a list is the script
then
  begin
    the first (function) of the list (procedure) is
       displayed_with_return;
    the first (function) of the list is the command,
    and the rest (function) of the list is the
    pending_script
```

```
  end_block;

If given a list is the pending_script, and
   something is not the command
then the list is the script;

end_rules.

!  activate the rules  !

The PI1_rules (procedure) are activated.
Nil is the current_node.
"PI-1 System loaded" (procedure) is
   displayed_with_return.
```

LIST OF REFERENCES

1. Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," CACM, vol. 21, No. 8, pp. 613-641, August 1978.

2. MacLennan, B. J., Principles of Programming Languages: Design, Evaluation, and Implementation, Holt, Rinehart, and Winston, 1983.

3. Winograd, Terry, "Beyond Programming Languages," CACM, Vol. 22, No. 7, pp. 391-400, July 1979.

4. Naval Postgraduate School Report NPS 52-83-001, A View of Object-Oriented Programming, by B. J. MacLennan, February 1983.

5. Dahl, O., and Nygaard, K., "SIMULA--An ALGOL-Based Simulation Language," CACM, Vol. 9, No. 9, pp. 671-678, September 1966.

6. Kay, A., "Microelectronics and the Personal Computer," Scientific American, Vol. 237, No. 3, pp. 230-244, September 1977.

7. Landin, P.J., "A Correspondence Between Algol 60 and Church's Lambda Notation," CACM, Vol. 8, No. 2, pp. 89-101, February 1965.

8. McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, I," CACM, Vol. 3, No. 4, pp. 185-195, August 1960.

9. Kowalski, R., Logic for Problem Solving, North-Holland, 1979.

10. Feigenbaum, Edward, and McCorduck, Pamela, The Fifth Generation, Addison-Wesley, 1983.

11. Shortliffe, E. H., Computer-Based Medical Consultations: MYCIN, American Elsevier, 1976.

12. Pople, Harry E., Jr., "On the Mechanization of Abductive Logic," Third International Joint Conference on Artificial Intelligence, Stanford, CA, 1973.

13. McDermott, John, "R1: A Rule-Based Configurer of Computer Systems," Artificial Intelligence, Vol. 19, No. 1, 1982.

14. McArthur, Heinz M., Design and Implementation of an Object-Oriented, Production-Rule Interpreter, MS Thesis, Naval Postgraduate School, Monterey, California, December 1984.

15. Naval Postgraduate School Report NPS 52-84-026, The Four Forms of Omega, by B. J. MacLennan, December 1984.

16. MacLennan, B. J., "Values and Objects in Programming Languages," SIGPLAN Notices, Vol. 17, No. 12, pp. 70-79, December 1982.

17. Stanford Computer Science Department Report No. CS-403, Hints on Programming Language Design, by C. A. R. Hoare, October 1973.

18. Borland International, Turbo Pascal--Reference Manual, Version 3.0, 1985.

19. Winston, Patrick Henry, Artificial Intelligence, Addison-Wesley, 1984.

20. Naval Postgraduate School Report NPS 52-85-006, Experience with Omega: Implementation of a Prototype Programming Environment, Part I, by B. J. MacLennan, May 1985.

BIBLIOGRAPHY

Barrett, William A., and Couch, John D., Compiler Construction: Theory and Practice, Science Research Associates, 1979.

Clockskin, W. F., and Mellish, C. S., Programming in Prolog, Springer-Verlag, 1981.

Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," CACM, Vol. 13, No. 6, June 1970.

Goldberg, A., and Ribson, D., Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.

Hopcroft, John E., and Ullman, Jeffrey D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

MacLennan, B. J., Functional Programming Methodology: Theory and Practice, to be published by Addison-Wesley.

Newell, A., and Simon, H., Human Problem Solving, Prentice-Hall, 1972.

Rich, E., Artificial Intelligence, McGraw-Hill, 1983.

Stanat, Donald F., and McAllister, David F., Discrete Mathematics in Computer Science, Prentice-Hall, 1977.

Warren, D. H., Pereira, L. M., and Pereira, F., "Prolog--The Language and Its Implementation Compared with LISP," SIGPLAN Notices, Vol. 12, No. 8, pp. 109-115, August 1977.

Winston, P. H., and Horn, B. K., LISP, Addison-Wesley, 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
4.	Computer Technology Programs (Code 37) Naval Postgraduate School Monterey, California 93943-5100	1
5.	Associate Professor Bruce J. MacLennan Code 52ML Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5100	1
6.	Captain Robert P. Ufford, USA 12 Lenape Lane Newark, Delaware 19713	2
7.	Dr. Robert Grafton Code 433 Office of Naval Research 800 N. Quincy Arlington, VA 22217-5000	1
8.	Professor Jack M. Wozencraft, 62Wz Department of Electrical and Comp. Engr. Naval Postgraduate School Monterey, CA 93943-5100	1
9.	Dr. Robert M. Balzer USC Information Sciences Inst. 4676 Admiralty Way Suite 10001 Marina del Rey, CA 90291	1
10.	Mr. Dennis Hall New York Videotext 104 Fifth Avenue, Second Floor New York, NY 10011	1
11.	Mr. A. Dain Samples Computer Science Division - EECS University of California at Berkeley Berkeley, CA 94720	1

215629

Thesis
U143
c.1

Ufford
The design analysis of a stylized hatural grammar for an object-oriented language (Omega).